SYBASE®

An SAP® Company

Authoring Reference Manual

Sybase Aleri Streaming Platform 3.2

DOCUMENT ID: DC01294-01-0320-02

LAST REVISED: December, 2010

Table of Contents

About This Guide

1. Related Documents

This guide is part of a set. The following list briefly describes each document in the set.

Product Overview	Introduces the Aleri Streaming Platform and related Aleri products.
Getting Started - the Aleri Studio	Provides the necessary information to start using the Aleri Studio for defining data models.
Release Bulletin	Describes the features, known issues and limitations of the latest Aleri Streaming Platform release.
Installation Guide	Provides instructions for installing and configuring the Streaming Processor and Aleri Studio, which collectively are called the Aleri Streaming Platform.
Authoring Guide	Provides detailed information about creating a data model in the Aleri Studio. Since this is a comprehensive guide, you should read the *Introduction to Data Modeling and the Aleri Studio*. first.
Authoring Reference	Provides detailed information about creating a data model for the Aleri Streaming Platform.
Guide to Programming Interfaces	Provides instructions and reference information for developers who want to use Aleri programming interfaces to create their own applications to work with the Aleri Streaming Platform. These interfaces include: • the Publish/Subscribe (Pub/Sub) Application Programming Interface (API) for Java • the Pub/Sub API for C++ • the Pub/Sub API for .NET • a proprietary Command & Control interface • an on-demand SQL query interface
Utilities Guide	Collects usage information (similar to UNIX® man pages) for all Aleri Streaming Platform command line tools.
Administrators Guide	Provides instructions for specific administrative tasks related to the Aleri Streaming Platform.
Introduction to Data Modeling and the Aleri Studio	Walks you through the process of building and testing an Aleri data model using the Aleri Studio.
SPLASH Tutorial	Introduces the SPLASH programming language and illustrates its capabilities through a series of examples.
Frequently Asked Questions	Answers some frequently asked questions about the Aleri Streaming Platform.

Chapter 1. Authoring Preliminaries

All three authoring methods — Aleri Studio, Aleri SQL, and AleriML — share some commonalities in names, types, and expressions.

1.1. Data Types and Literal Constants

The Sybase® Aleri Streaming Platform has the following set of primitive data types:

int32	32-bit integers
int64	64-bit integers
money	Fixed-point numbers, with a default precision of 4 decimal digits.
double	Double precision floating-point numbers.
date	Date and time values, with one second precision.
timestamp	Date and time values, with one millisecond precision.
string	Character strings of arbitrary length.

More information about data types can be found in Appendix B, *Data Types, Operators and Functions*.

Constants have the following types:

- Numbers without a decimal point (for example, -101, 8, +93734) have type int32.
- Decimal numbers (for example, -172.76245, 186.756) have type double.
- Decimal numbers with a trailing “d” or “D” (for example, -172.7624d, 186.756D) have type money.
- Character strings in single quotes (for example, 'this is a string') are of type string. Character strings can contain the following escape sequences:
  - \b (for backspace)
  - \n (for line feed)
  - \r (for carriage return)
  - \t (for tab)
  - \ddd (for octal values of at most three digits, as in \013)
  - \xhh (for hexadecimal values of at most two digits, as in \x1a)

1.2. Names

The name of any Sybase Aleri Streaming Platform object must adhere to the following rules:

- A name is either a sequence of alphabetic characters, digits, and underscore characters, or a sequence of any characters enclosed in double quotes.

- If a name is not enclosed in double quotes, it must begin with an alphabetic character or an underscore.
- A name cannot contain spaces unless it is enclosed in double quotes.
- A name cannot be a Reserved Word unless it is enclosed in double quotes. Reserved words are case insensitive, so for example, a name cannot be "AND" or "and" or "AnD". See Appendix A, *Reserved Words* for the list of Reserved Words.
- Columns cannot be named "rowid" or "rowtime".

Note:

Double quotes in AleriML must be written with the XML escape sequence "&quot".

1.3. Expressions

Expressions tell how to compute a value from other values. They can be as simple as the expressions `1` and `2+2`, or as complex as you like.

The Sybase Aleri Streaming Platform provides a number of built-in operators and functions for performing complex calculations, as well as means of defining and calling your own functions. For a complete list of internal functions and operators, see Appendix B, *Data Types, Operators and Functions*.

Parentheses can be used to group expressions and change the order of operator precedence. For instance, `(9+1)*7` is an expression that computes to 70, whereas `9+1*7` computes to 63.

References to columns in streams use the standard dot notation. For instance, to refer to column "Price" in a stream called "Trades", you write `Trades.Price`. Variables can also be used within expressions. For instance, if there is a variable named "scale", you could write the expression `scale * Trades.price`.

There's also an expression for setting a variable, and for combining such expressions by sequencing. The expression `v := 9` sets the value of the variable `v` to 9, and returns that new value 9. Sequencing is done with parentheses and semicolon. For instance, `(v := 8; print(string(v)); 7)` returns the value 7, but sets `v` to 8 and prints the value of `v` before doing so.

1.4. Notational Conventions

The following conventions are used to describe Aleri SQL and AleriML:

- Square brackets [ ] represent optional elements.
- Curly brackets { } represent required elements where there is a choice of which element to use.
- The logical or symbol (|) separates choices within curly brackets { }.
- Asterisk (*) indicates that an element may not be present, or may repeat any number of times. Plus (+) indicates that an element must appear and may repeat any number of times.

Chapter 2. Authoring in SQL

Aleri SQL is one of three supported tools for creating data models that run on the Sybase Aleri Streaming Platform. Aleri SQL provides a familiar environment for those with experience writing queries in SQL since it is based on the ANSI SQL99 standard, with extensions for working with streaming data.

Aleri SQL also has the advantage of being the most concise way of expressing a data model. Setting a data model up in the Aleri Studio can be more time consuming, due to the nature of visual development paradigms, and AleriML is more verbose. Therefore, even someone new to SQL may find it the most efficient authoring environment.

2.1. Aleri SQL Overview

In order to conform with standard SQL, Aleri SQL maps stream processing elements to standard SQL constructs. Streams are mapped to tables and views. Aleri SQL uses the following basic SQL constructs to build all the necessary underlying elements of the data model:

`Create Store`	to define a storage manager
`Create Table`	to define a source (input) stream
`Create Materialized View`	to define a continuous query that produces a derived stream (other than a FlexStream or Pattern Stream)
Create Program View	to define a FlexStream.
Create Pattern View	to define a Pattern Stream.
Create Data Location	to define a Data Location element.
Create Connection	to define a Connection element
Create Connection Group	to define a startup group
`Create Module, Create Cluster`	to implement a distributed model
`Declare`	to define parameters that can be used in expressions
Create Function	to define native and external functions.
`Grant`	to set access control

See the *Authoring Guide* for an explanation of the different elements that make up a data model.

Note

To conform with standard SQL, “Tables” represent source streams and “Views” represent derived streams. In this section of the guide, tables are often referred to as “source streams” and views (both materialized views and program views) are often referred to as “derived streams”. Collectively, tables and views may be referred to as “streams”. The use of the word “stream” conveys the notion that these are not static data sets but have data flowing through them at all times.

2.2. Store Definition

A store defines the physical storage characteristics for the streams (tables, materialized views, and program views) assigned to it. Every table, materialized view and program view must be assigned to a store. There are three types of stores: Log Store, Memory Store and Stateless Store. Log Stores are persistent — they guarantee data state recovery after failure, since all data is logged to disk. Memory Stores provide data retention with higher performance than the log store, but they are not persistent — all data is held in memory. Stateless Stores can be viewed as transient Stores that do not provide data retention. See the *Authoring Guide* for a more complete description of the differences between the different store types.

The syntax for defining a Store in Aleri SQL is as follows:

```
CREATE STORE StoreName
{ LOGSTORE ON Location
[ SYNC is { true | false } ] | MEMSTORE | STATELESSSTORE }
[ MAXSIZE [ IS ] SizeInMb ] , [ INDEX [ IS ] { HASH | [(TREE)]} ] [ ; ]
```

where:

- `StoreName` is the unique name of the Store which must conform to the naming conventions.
- *LOGSTORE | MEMSTORE | STATELESSSTORE* (required) specifies the type of storage manager to be created.
- *Location* is the file path where Log Stores are persisted to disk.

  This is mandatory for Log Stores. It is ignored for Memory Stores and Stateless Stores.
- **SYNC** (optional) specifies whether the persisted data is updated synchronously with every stream being updated, or whether it can be updated asynchronously.

  Setting the *sync* value to `true` guarantees that every record acknowledged by the system is persisted at the expense of performance.

  Setting the value to `false` improves performance, but it does not guarantee a prevention of data loss. Data could be lost in the case of a hard system shutdown if stream data has not been flushed to the disk yet.

  This property is only meaningful for log Stores. The default value is false.
- *MAXSIZE* (optional) specifies the maximum size of the LOGSTORE in MB. This translates to the fullsize attribute in AleriML. If this option is not specified, it defaults to 8 MB. When it is specified for other types of stores, the Sybase Aleri Streaming Platform ignores it. If the maximum size is reached the server will shut down.
- *INDEX* (optional) specifies the kind of indexing to be used: *HASH* or *TREE*. This is optional and only applies to Memory Stores. The default value is *TREE*.

2.3. Source Streams

A source stream is a stream that receives data from an external source. There can be any number of source streams in a data model. All data flowing into a stream must have the same structure: each record must have the same set of fields. Although all input channels are called “streams”, data does not actually have to arrive as a stream; Source streams place no restriction on how frequently or infrequently data arrives. Therefore, static reference data that is loaded from a file or a database is still loaded into the model via a stream.

The SQL `CREATE TABLE` statement is used to define a source stream. The Aleri SQL translator will create an underlying SourceStream depending on the properties set in the `CREATE TABLE` statement.

A source stream is defined using the following SQL syntax:

```
CREATE TABLE TableName [ FOR INSERT ] (
ColumnName Type [ ,...n ]
[,PRIMARY KEY ( KeyColumn [ ,...n ] )
] [ STORE IS ] StoreName
[[ [ ,RETAIN ] { [ Duration { SEC | MIN | HRS | HOUR | DAYS } |(ALWAYS) ] } ]
| [ ,MAX RECORDS NoOfRecords [ :SlackValue ] ]]
[ ,EXPIRES IN ExpiryDuration { SEC | MIN | HRS | HOUR | DAYS } [ FROM ExpireFrom-
Column ]
SET ExpiryColumn [ NoOfTimes TIMES ] ]
[ ,AUTO GENERATE AutoGenColumn ]
[ WHERE FilterExpression [ { AND | OR } ...] \) ] [ ; ]
```

Note:

If a `WHERE` clause or the `FOR INSERT` keywords are specified, the Table is marked as insert only.

where:

- *`TableName`* (required) is the name for the Base Stream being defined. The name must follow the naming conventions that are specified in Section 1.2, “Names”.
- **FOR INSERT** (optional) directs the translator to generate an insert only table. This option is recommended when you know that a table will only receive inserts. This not only improves performance, but it also allows you to use this stream in INNER JOINS.
- *`ColumnName`* (required) is the name of the column that is being defined. This name must follow the naming conventions and be unique within the stream.
- *`Type`* is one of the supported data types. Refer to Section B.1, “Data Types” for a list of supported types.
- **PRIMARY KEY** (required) specifies one or more columns on which the table will be keyed. *`KeyColumn`* is the name of a column. A primary key can contain one or more non-null columns. A primary key must be defined for every table.
- **STORE** *`StoreName`* (required) specifies which storage manager is to be used. *`StoreName`* is the name assigned to the storage manager in the `CREATE TABLE` statement. The persistence properties of the stream are determined by the type of assigned store.
- **RETAIN** *`Duration`* (optional) defines the retention period for the stream. The value *`ALWAYS`* means that all data is retained. *`Duration`* is an integer that represents the number of seconds, minutes, hours, or days for which the data is to be retained. Either `RETAIN` or `MAX RECORDS` (below) may be specified. Retention periods will be ignored for tables if the `INSERT ONLY`, `WHERE` or AUTOGEN clause is specified.
- **MAX RECORDS** *`NoOfRecords :SlackValue`* (optional) specifies retention based on number of records rather than time. *`NoOfRecords`* is an integer that represents the maximum number of records to be retained. If there are excess records, the older records are deleted. *`SlackValue`* is an optional parameter that specifies the number of records the stream can have over the defined max *`NoOfRecords`* parameter before old records are purged. This improves efficiency by by avoiding a purge on every incoming record. By default, this value is set to 10% of the *`NoOfRecords`* parameter. `MAX RECORDS` or `RETAIN` (above) may be specified, but not both. Retention policies can only be defined for non-insert only tables. If `INSERT ONLY`, `WHERE`, or `AUTOGEN` are specified, the `MAX RECORDS` value will be ignored.

- **EXPIRES IN** (optional) specifies that a flag will be set on each record after the specified `Expiry-Duration` time period has elapsed. The column to be used for the expiry flag is specified by `Ex-piryColumn` and the value in the column is incremented when the expiry time period has elapsed. The expired flag may be incremented more than once by specifying a value greater than one for the optional `NoOfTimes` parameter. By default, the expiry period is the time since the record was last updated (or created if it has never been updated); alternatively, the expiry period can be the time since the timestamp contained in a column specified with `ExpireFromColumn`.

- `ExpiryDuration` specifies the time period after which the expiry flag on a record is set. The time period is relative to the time the record was last updated unless an alternate time reference is specified in the `ExpiryFromColumn`. This must be a positive integer greater than 0.

- `ExpireFromColumn` is an optional parameter which specifies that the Expiry Duration is relative to the value in this column rather than the last time the row was updated or deleted. The specified column must be defined as part of the current stream and must be of type date.

- `ExpiryColumn` specifies the name of the column in the current table that will be used to hold the expiry flag. It will be this column whose value is incremented (updated) in order to indicate that the record has expired. This column must be of type integer.

- `NoOfTimes` is an optional parameter specifying the maximum number of times the expiry flag may be incremented. When the max is reached, the flag will no longer be incremented until it is reset to zero upon an update or new value in the ExpiryFromColumn. The default value is 1.

- **WHERE** (optional) specifies a filter to be applied to the input stream. In other words, any record that does not pass the filter is not passed down to any dependent derived streams. The inclusion of the `WHERE` clause causes this stream to be eligible only for insert-only operations.

- `filterExpression` The filter expression to be applied. Note that when referring to a column from the view being defined prefix the column name with a '.'. For example .column1. For information on defining a filter expression see Section 2.4, “Continuous Queries”.

- **AUTO GENERATE** (optional) specifies that an automatically generated sequence number should be added to each record. The column will be filled in with an increasing sequence of integers starting with 0. The column can be used as a key, or part of a key, in the *keys* attribute. Therefore, the *autogen* feature is particularly useful for streaming data that has no natural primary key, but where each record is regarded as an insert. This option can only be used with Auto Streams (insert-only) streams and ignored for regular streams.

- *AutoGenColumn* specifies the name of the column that will contain an automatically generated sequence number. The column must be of type int64. Only one column per stream can be auto generated.

The following example shows the table definition for a simple store.

```
// This is a simple store used in all the following examples
CREATE STORE "store1" MEMSTORE

/************************************************************
  The following is an example of the most basic Table
  definition with no options.
  This generates a SourceStream primitive in AleriML.
**********************************************************/

CREATE TABLE Table1 (
    KeyColumn int32, Column1 string, Column2 date,
  PRIMARY KEY (KeyColumn), STORE IS "store1" );
```

```
/*******************************************************
  The following is an example of a simple Insert Only Table
  with no options.
  This statement generates a SourceStream primitive in AleriML.
*********************************************************/

CREATE TABLE Table2 FOR INSERT (
    KeyColumn string, Column1 int32, Column2 int64,
  PRIMARY KEY (KeyColumn), STORE IS "store1" );

/*********************************************************
  The following is an example of a Table definition with
  all supported options.

  This generates a SourceStream primitive in AleriML
  The WHERE and AUTO GENERATE clauses are only supported
  for Insert Only Streams.
  The MAX RECORDS clause can be used instead of the RETAIN
  clause used in this example for record-based retention.

************************************************************/

CREATE TABLE Table3 (
    KeyColumn1 int64, KeyColumn2 string, Column1 money,
    Column2 int64, Column3 string, Column4 date, Column5 int32,
  PRIMARY KEY (KeyColumn1, KeyColumn2),
  STORE IS "store1",
  RETAIN 10 SEC,
  // The EXPIRES clause causes the value in Column5 to be set to 1
  // after the first 10 Minutes of no activity on the record from
  // the time specified in Column4. Thereafter the value in Column5
  // is incremented by 1 after every 10 minutes of no activity
  // 4 more times. If there is activity in the record then Column5
  // is reset to 0 and the process begins again.
  EXPIRES IN 10 MIN FROM Column4 SET Column5 5 TIMES );

/*******************************************************
  The following is an example of an Insert Only Table with
  all possible options.

  This statement generates an SourceStream primitive in XML with the
  insert-only flag turned on. Insert Only Streams cannot have Expiry
  and Retention Clauses because these generate updates. When the AUTO
  GENERATE option is specified, a Table can have only have a single
  key column and it must be of type int64. Note how the columns in
  the filter clause are prefixed with a '.'. This is required when
  referencing columns in the Table/View being defined.
********************************************************/

CREATE TABLE Table4 FOR INSERT (
    KeyColumn int64, Column1 int32, Column2 int64,
  PRIMARY KEY (KeyColumn), STORE IS "store1",
  AUTO GENERATE KeyColumn
  WHERE .Column2 > 10 AND .Column1 > 100 );
```

2.4. Continuous Queries

The Continuous Query takes one or more other streams (tables, materialized views, or program views) as input, and produces the result as a derived stream or, in SQL terminology, a materialized view. The SQL `CREATE MATERIALIZED VIEW` statement defines a Continuous Query that produces a derived stream. The Aleri SQL translator will create one or more underlying derived stream elements to generate the materialized view.

The following is the Aleri SQL syntax to define a continuous query:

```
CREATE [MATERIALIZED] VIEW ViewName
PRIMARY KEY ( KeyColumn [,...])
STORE StoreName
[ [ [ INTERMEDIATE STORE [ IS ] StoreName ]
[ | EXPIRES IN ExpiryDuration { SEC | MIN | HRS | HOUR | DAYS } [ FROM ExpireFrom-
Column ]
SET ExpiryColumn [ NoOfTimes TIMES ] ]
| [ DECLARE VariableName [ eventCache( CacheDefinition ) | VariableType ] ]
] | [ CREATE FUNCTION FunctionName { FunctionDefinition } RETURN ReturnType ]
[ ... ] ]
AS SELECT
Expr [ [ AS ] ColumnName ] [ ,Expr ...]
FROM
{ TableName | ViewName } [ AliasName ]
[ RETAIN { [[ Duration { SEC | MIN | HRS | HOUR | DAYS } ]] |(ALWAYS) } |
[ MAX RECORDS NoOfRecords [ :SlackValue ] ] ]
[ Join Type { TableName
| ViewName } ] [ AliasName ]
[ RETAIN { [[ Duration { SEC | MIN | HRS | HOUR | DAYS } ]] |(ALWAYS) } |
[ MAX RECORDS NoOfRecords [ :SlackValue ] ] ] ON
LeftExpression = RightExpression [ AND LeftExpression ...]
[ Join Type ...]
[ WHERE FilterExpression [ AND | OR ... ...] ]
[ GROUP BY GroupExpression [ , ...] ]
[ GROUP ORDER BY OrderExpression [ (ASC) | DESC ] [ , ...] ]
[ GROUP HAVING GroupHavingExpression [ AND | OR ...] ]
[ HAVING HavingExpression [ AND | OR ] ]
[ UNION SELECT... ...] [ ; ]
```

Components:

- `ViewName` (required) is the name for the Materialized View being defined. The name must be unique and should follow the naming conventions specified in Section 1.2, “Names”.
- **PRIMARY KEY** `KeyColumn` (required) defines the primary key for a derived stream. A primary key can be made up of one or more columns that are defined to be non-null. `KeyColumn` is set to a column name in the view as defined in the Select statement. A primary key must be defined for every derived stream.
- **STORE** *StoreName* (required) specifies the Store to be used. The persistence properties for the stream are derived from the type of store specified.
- **INTERMEDIATE STORE** (optional) specifies the store that will be used by any intermediate streams required to build this materialized view. If not specified, it defaults to the STORE value. Note that there is no direct equivalent to this property in AleriML. This is only used to specify to the translator to use a different store for intermediate streams rather than the store for the stream being defined. This option is typically used for derived streams that use Log Stores because it gives the ability to store the intermediate streams in a Memory Store or Stateless Store to improve performance.
- **RETAIN** (optional) specifies the minimum amount of time the Sybase Aleri Streaming Platform retains the data in the view before archiving or purging it. The duration can be specified in minutes, hours, or seconds. If this parameter and the *`MAX RECORDS`* parameter are not specified, the data in the view is always retained. Note that this property can only be specified for a materialized view that is a direct copy of the input table/view (in other words, the column order and the number of columns

of the view must match that of the input specified in `SELECT...FROM...`, and there can be no filter or group by clauses). See the description of this property in Section 2.3, "Source Streams" for more information.

- **MAX RECORDS** (optional) specifies the maximum number of records to keep in a derived stream. When the number of records exceeds this maximum number, the older records are eligible for purging or archiving. Like the RETAIN property, currently, this property can only be specified for materialized views that are direct copies of the input table or view. See the description of this property in Section 2.3, "Source Streams" for more information.
- **EXPIRES IN** specifies that a flag will be set on each record after the specified *`ExpiryDuration`* time period has elapsed. The column to be used for the expiry flag is specified by *`ExpiryColumn`* and the value in the column is incremented when the expiry time period has elapsed. The expired flag may be incremented more than once by specifying a value greater than one for the optional *`NoOfTimes`* parameter. By default, the expiry period is the time since the record was last updated (or created if it has never been updated); alternatively the expiry period can be the time since the timestamp contained in a column specified with *`ExpireFromColumn`*.
- *`ExpiryDuration`* specifies the time period after the expiry flag on a record is set. The time period is relative to the time the record was last updated unless an alternate time reference is specified in the *`ExpiryFromColumn`*. This must be a positive integer greater than 0.
- *`ExpireFromColumn`* is an optional parameter that specifies that the Expiry Duration is relative to the value in this column rather than the last time the row was updated or deleted. The specified column must be defined as part of the current stream and type date.
- *`ExpiryColumn`* specifies the name of the column in the current table that will be used to hold the expiry flag. This column's value will be incremented (updated) to indicate that the record has expired. This column must be of type integer.
- *`NoOfTimes`* is an optional parameter specifying the maximum number of times the expiry flag may be incremented. When the max is reached, the flag will no longer be incremented until it is reset to zero upon an update or new value in the ExpiryFromColumn. The default value is 1.
- **DECLARE** (optional) defines one or more variables and functions that are local to the view being declared. The variable can be one of the basic types or an eventCache. There may be any number of local variables and functions defined.
- *`VariableName`* is the name of the variable and must follow the standard naming conventions. Refer to Section 1.2, "Names" for more information. The variable names must be unique with the list of variables defined for the current view.
- *`VariableType`* is the data type of the variable being defined. The data type must be a supported data types or an eventCache. Refer to Section B.1, "Data Types" for a list of supported types.
- *`CacheDefinition`* is the definition for the event cache. Refer to Section 4.3.7, "Event Caches" for more information on defining and using event aches.

  ### Note:

  Although the syntax allows a user to specify more than one `INTERMEDIATE STORE` and `EXPIRES` clause for a derived stream, only the last is honored by the translator. The translator generates a warning when any clause is repeated.

- *`FunctionName`* is the name of the function being defined. The function name must be unique within the scope of the view being defined and must follow the standard naming conventions specified in the section Section 1.2, "Names".

- `FunctionDefinition` is the body of the function and is defined using the SPLASH syntax. Refer to the section Section 4.5, "Functions" for more information on using functions and defining the body of the function.

- `ReturnType` defines the type of the function return value. The type can one of the basic types described in section Section B.1, "Data Types".

- **AS SELECT** [`Expr` **AS** `ColumnName`] (required) defines the columns of the materialized view (that is, the structure of the output records for this stream). `Expr` is an expression that evaluates to a scalar value. An expression can be a simple reference to a value from the input table/view (specified in the `FROM` clause), a formula to calculate a new value from the column in the view, or even a constant. It could also be a complex expression consisting of one or more internal/external function calls, if-then-else-end statements, and so on. Refer to Section 1.3, "Expressions" for more information. If the expression is a direct copy of an input column, the ColumnName can be omitted, in which case the name will match the name of the input column. Each `ColumnName` must be unique within the view.

- **FROM** `TableName` | `ViewName` [`AliasName`] (required)specifies one or more streams that will be the input(s) to the materialized view being defined. There must be at least one input stream. An optional `AliasName` can be specified to use when referring to the input stream in subsequent expressions. If the Alias is not explicitly specified, the stream name is used in expressions. The Alias Name must follow the proper naming conventions (seeSection 1.2, "Names").

- *JoinType* (optional) specifies multiple input streams in the `FROM` clause. This can be one of [`INNER`] `JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN`. Each Join Type is followed by an `ON` clause, which specifies the relationship between the tables. See Section 2.14.1, "Join Expressions" for more information.

- **WHERE** optionally specifies one or more Filter Expressions that evaluate to `true` or `false` and are concatenated to each other by either an **AND** or **OR** operator. Although both sides of the expression can be constants, typically at least one side of the filter clause refers to one or more columns in the input stream(s). See Section 2.14.2, "Filter Expressions" for more information.

- **GROUP BY** `GroupExpression` optionally specifies values on which the data is to be grouped. If a **GROUP BY** clause is used, at least one `GroupExpression` must be specified. The expression must be based on a column in one of the input streams. There must be one `GroupExpression` for each key column in the materialized view that is being defined, and each `GroupExpression` must match the `Expr` for that column in the **SELECT** clause.

- **GROUP ORDER BY** `OrderExpression` can optionally be used in conjunction with the **GROUP BY** clause to order the rows in each group. `OrderExpression` is any expression not containing aggregation operations like `MIN` and `MAX`. It can also be followed by the keyword ASC or DESC for ascending or descending order. If `ASC/DESC` is not specified, it defaults to ascending. One or more Order expressions may be specified, separated by commas. This ordered group is used in functions like `RANK`, `FIRST`, and `LAST`. See Appendix B, *Data Types, Operators and Functions* for more information.

- **GROUP HAVING** `GroupHavingExpression` (optional) is like a **WHERE** clause for groups that filter out records from the group before applying aggregation operations. This clause can only be specified in conjunction with the **GROUP BY** clause. The `GroupHavingExpression` is identical to a filter expression, except that it is used only when there is aggregation to filter rows in a group before the aggregation operations are applied. This expression must follow all the rules of a filter expression as described in Section 2.14.2, "Filter Expressions".

- **HAVING** `HavingExpression` (optional) is a clause similar to the **WHERE** clause. The main difference is that the filter is executed after all the rules have been applied. It can have aggregation clauses and it can refer to columns within the current table. Like the `WHERE` clause, there may be one or more filters that are concatenated by `AND/OR` operators.

- **UNION SELECT** (optional) allows you to perform a `UNION` on the results of multiple `SELECT` statements. The structure of each `SELECT` statement on which the `UNION` is to be applied must be the same.

2.4.1. Examples of Materialized View Definitions

We use the following tables as inputs to the examples below.

```
CREATE STORE "store1" MEMSTORE;
CREATE STORE store2 LOGSTORE ON 'store2' MAXSIZE 16;
CREATE STORE store3 MEMSTORE;
CREATE TABLE Table1 (
    KeyColumn1 int32, KeyColumn2 string, Column1 string, Column2 date,
    Column3 money,
  PRIMARY KEY (KeyColumn1,KeyColumn2),
  STORE IS "store1" );
CREATE TABLE Table2 (
    KeyColumn1 int32, Column1 string, Column2 date,
  PRIMARY KEY (KeyColumn1),
  STORE IS "store1" );
CREATE TABLE Table3 (
    KeyColumn1 int32, KeyColumn2 string, Column1 string, Column2 date,
    Column3 money,
  PRIMARY KEY (KeyColumn1,KeyColumn2),
  STORE IS "store1" );
```

The following is an example of the most basic Materialized View definition with no options. It generates a Compute Stream primitive in AleriML.

```
CREATE MATERIALIZED VIEW View1
  PRIMARY KEY (KeyColumn1)
  STORE IS "store1"
  AS SELECT tbl.KeyColumn1, tbl.Column1, tbl.Column2 FROM Table2 tbl ;
```

The following is an example of a Materialized View definition that generates a Compute Stream primitive in AleriML and contains all possible options.

Notes:

- This generates a Compute Stream primitive in AleriML.
- The `MAX RECORDS` clause after *`Table1`*, *`lstore1`* and *`lstore2`* below can be replaced with the `RETAIN` clause to specify retention by time instead of by records.
- When the `EXPIRES` clause is used, neither the `STORE` nor the `INTERMEDIATE STORE` can be set to a log store.

```
CREATE MATERIALIZED VIEW View_1a
  PRIMARY KEY (KeyColumn1, KeyColumn2)
  STORE IS "store1"
  INTERMEDIATE STORE "store1"
  EXPIRES IN 10 SEC
  SET ExpiryColumn 3 TIMES
  AS SELECT tbl.KeyColumn1, tbl.KeyColumn2, tbl.Column1,
       tbl.Column2, 0 ExpiryColumn,
       // Determines the maximum Column1 value in the last 5 records.
       Sequence(aggregate(insert, lstore1, string, tbl.Column1),
       aggregate(max, \ lstore1, string, -1)) Moving_Max_Column1,
       // Determines the maximum Column2 value in the last 5 records.
       Sequence(aggregate(insert, lstore2, date, tbl.Column2),
       aggregate(min, \ lstore2, date, -1)) Moving_Min_Column2
    FROM Table1 tbl MAX RECORDS 20 ;
```

This statement generates a Filter Stream primitive in AleriML. If the columns in the Materialized View are not identical to the input, then a Filter Stream will be generated followed by a Compute Stream to match the specified column definition.

```
CREATE MATERIALIZED VIEW View2
  PRIMARY KEY (KeyColumn1) STORE IS "store1"
  AS SELECT * FROM  Table2 tbl
     WHERE tbl.Column1 in ('Chicago', 'London', 'Frankfurt');
```

This statement generates an Aggregate Stream primitive in AleriML.

```
CREATE MATERIALIZED VIEW View3
  PRIMARY KEY (KeyColumn1) STORE IS "store1"
  AS SELECT tbl.KeyColumn1, MAX(tbl.Column1) Max_Column1,
      MIN(tbl.Column2) Min_Column2 FROM Table1 tbl
     GROUP BY tbl.KeyColumn1;
```

This statement generates a Filter Stream Primitive for the `WHERE` clause and an Aggregate Stream primitive in AleriML followed by a Compute Stream primitive to fulfill the `Having` Clause.

Notes:

- When the `EXPIRES` clause is used, neither the `STORE` nor the `INTERMEDIATE STORE` can be set to a log store.
- It is more efficient to use the `GROUP HAVING` clause instead of the `WHERE` clause. In this example, it will be more efficient to combine the `WHERE` and `GROUP HAVING` clauses into a single `GROUP HAVING` clause.
- The `MAX RECORDS` clause can be replaced with the `RETAIN` clause to specify time based retention.
- The `LOCALSTORAGE` clause cannot be specified with an aggregation.
- One can do a join by specifying more than one table in the `FROM` clause. If this is done, however, a retention clause CANNOT be specified.

```
CREATE MATERIALIZED VIEW View_3a
  PRIMARY KEY (KeyColumn1)
  STORE IS "store1"
  INTERMEDIATE STORE IS "store3"
  EXPIRES IN 10 MINUTES SET ExpiryColumn 3 TIMES
  AS SELECT tbl.KeyColumn1, MAX(tbl.Column1) Max_Column1,
       MIN(tbl.Column2) Min_Column2,
       FIRST(tbl.Column1) First_Column1,
       LAST(tbl.Column3) Last_Column3,
       0 ExpiryColumn
     FROM Table1 tbl MAX RECORDS 20:10
     WHERE tbl.Column1 IN ('Chicago', 'London')
     GROUP BY tbl.KeyColumn1
     GROUP ORDER BY tbl.Column2
     GROUP HAVING tbl.Column3 > 100.0
     HAVING LAST(tbl.Column3) > 1000.0;
```

This statement generates a Join Stream primitive in AleriML:

```
CREATE MATERIALIZED VIEW View4
  PRIMARY KEY (KeyColumn1, KeyColumn2)
  STORE IS "store1"
  AS SELECT tbl1.*, tbl2.Column1 Table2_Column1
     FROM Table1 tbl1
       LEFT JOIN Table2 tbl2 ON (tbl1.KeyColumn1 = tbl2.KeyColumn1);
```

The following statement generates a Join Stream primitive in AleriML.

Notes:

- The following clauses can be used with the Joins: `WHERE GROUP BY`, `GROUP ORDER` (requires `GROUP BY`), and `GROUP HAVING` (requires `GROUP BY`).
- Retention cannot be specified when performing a join.
- A `LOCALSTORAGE` clause cannot be specified with a join.

```
CREATE MATERIALIZED VIEW View_4a
  PRIMARY KEY (KeyColumn1, KeyColumn2)
  STORE IS "store1"
  EXPIRES IN 5 SEC SET ExpiryColumn
  AS SELECT tbl1.*, tbl2.Column1 Table2_Column1, 0 ExpiryColumn
     FROM Table1 tbl1
       LEFT JOIN Table2 tbl2 ON (tbl1.KeyColumn1 = tbl2.KeyColumn1);
```

The following is an example of a Materialized View with a `UNION` operation with no options.

Notes:

- This statement generates a Union Stream primitive in AleriML.
- If an individual statement that makes up the Union Stream is not a direct copy of a source table/view, then intermediate Streams will be generated in addition to the Union Stream.
- The options that can be included in the individual SQL statements follow the same rules as mentioned in the previous examples.
- The `LOCALSTORAGE` clause cannot be specified in a `UNION` statement.

```
CREATE MATERIALIZED VIEW View5
```

```
  PRIMARY KEY (KeyColumn1, KeyColumn2)
  STORE IS "store1"
  AS SELECT * FROM Table1 UNION SELECT * FROM Table3;
```

The following is an example of a Materialized View with a `UNION` operation and all possible options.

Notes:

- This statement generates a Union Stream primitive in AleriML.
- If a individual statement that makes up the Union is not a direct copy of a source table/view, then intermediate Streams will be generated.
- The options that can be included in the individual SQL statements follow the same rules as mentioned in the previous examples.
- The `LOCALSTORAGE` clause cannot be specified in a `UNION` statement.

```
CREATE MATERIALIZED VIEW View_5a
  PRIMARY KEY (KeyColumn1, KeyColumn2) STORE IS "store1"
  EXPIRES IN 5 SEC SET ExpiryColumn
  AS SELECT *, 0 ExpiryColumn FROM Table1 RETAIN 1 HOUR UNION
     SELECT *, 0 ExpiryColumn FROM Table3 RETAIN 10 MINUTES ;
```

The following is an example of a Materialized View that copies the data from another stream directly and optionally specifies a retention policy on the data.

Notes:

- This statement generates a Copy Stream primitive in AleriML.
- In place of the '*' one can specify all the column names from the source stream in exactly the same order it appears in the Source. A '*' is just a convenient way representing it.
- If the source Stream is Materialized View and it is in a Log Store, this is the only way to specify retention on the data.
- Specifying any other option other than *`Retention`* will not result in a Copy Stream being generated.

```
CREATE MATERIALIZED VIEW View6
  PRIMARY KEY (KeyColumn1, KeyColumn2)
  STORE IS "store1"
  AS SELECT * FROM Table1 MAX RECORDS 10:5 ;
```

2.5. Program View Definition

A Program View is an Aleri SQL construct that is used to define a FlexStream element in the data model. As the name indicates, the `PROGRAM` is a set of methods written in SPLASH (see Chapter 4, *SPLASH Programming Language*) that defines the contents of the view. A Program View is a form of a Materialized View and can be used wherever a Materialized View is used. The syntax to declare a Program View:

`CREATE PROGRAM VIEW` *`ViewName`*

PRIMARY KEY (*`KeyColumn`* [,...])
STORE *`StoreName`* [[
[| EXPIRES IN *`ExpiryDuration`* {SEC | MIN | HRS | HOUR | DAYS} [FROM *`ExpireFrom-Column`*]
SET *`ExpiryColumn`* [*`NoOfTimes`* TIMES]]
| [DECLARE *`VariableName`* [eventCache(*`CacheDefinition`*) | *`VariableType`*]]
] | [CREATE FUNCTION *`FunctionName`* {*`FunctionDefinition`*} RETURN *`ReturnType`*]
[...]]
AS
`ColumnName Type` [,...n]
FROM
{ *`TableName`* | *`ViewName`* } [,...]
PROGRAM
[*`VariableName VariableType ;`* [...]]
{ [*`MethodName`* FOR] *`TableName`* { *`SplashProgram`* } [...] }

where

- *`ViewName`* is the name of the Program View being defined. The name must be unique and must follow the naming conventions specified in Section 1.2, “Names”.
- **PRIMARY KEY** *`KeyColumn`* specifies one or more columns that will form the key for this view. A primary key must be defined for every Program View.
- **STORE** *`StoreName`* is the name of Store that will be used to hold retained data for this view. The persistence properties for the view are derived from the type of Store specified.
- **EXPIRES IN** specifies that a flag will be set on each record after the specified *`ExpiryDuration`* time period has elapsed. The column to be used for the expiry flag is specified by *`ExpiryColumn`* and the value in the column is incremented when the expiry time period has elapsed. The expired flag may be incremented more than once by specifying a value greater than one for the optional *`NoOfTimes`* parameter. By default, the expiry period is the time since the record was last updated (or created if it has never been updated); alternatively the expiry period can be the time since the timestamp contained in a column specified with *`ExpireFromColumn.`*
- *`ExpiryDuration`* specifies the time period after the expiry flag on a record is set. The time period is relative to the time the record was last updated unless an alternate time reference is specified in the *`ExpiryFromColumn`*. This must be a positive integer greater than 0.
- *`ExpireFromColumn`* is an optional parameter that specifies that the Expiry Duration is relative to the value in this column rather than the last time the row was updated or deleted. The specified column must be defined as part of the current stream and type date.
- *`ExpiryColumn`* specifies the name of the column in the current table that will be used to hold the expiry flag. This column's value will be incremented (updated) to indicate that the record has expired. This column must be of type integer.
- *`NoOfTimes`* is an optional parameter specifying the maximum number of times the expiry flag may be incremented. When the max is reached, the flag will no longer be incremented until it is reset to zero upon an update or new value in the ExpiryFromColumn. The default value is 1.
- **DECLARE** (optional) defines one or more variables and functions that are local to the view being defined. The variable can be one of the basic types or an eventCache. There may be any number of local variables and functions defined.
- *`VariableName`* is the name of the variable and must follow the standard naming conventions. Refer to Section 1.2, “Names” for more information. The variable names must be unique with the list of variables defined for the current view.

- `VariableType` is the data type of the variable being defined. The data type must be a supported data types or an eventCache. Refer to Section B.1, "Data Types" for a list of supported types.

- `CacheDefinition` is the definition for the event cache. Refer to Section 4.3.7, "Event Caches" for more information on defining and using even aches.

  Note:

  Although the syntax allows a user to specify more than one `INTERMEDIATE STORE` and `EXPIRES` clause for a derived stream, only the last is honored by the translator. The translator generates a warning when any clause is repeated.

- `FunctionName` is the name of the function being defined. The function name must be unique within the scope of the view being defined and must follow the standard naming conventions specified in the section Section 1.2, "Names".

- `FunctionDefinition` is the body of the function and is defined using the SPLASH syntax. Refer to the section Section 4.5, "Functions" for more information on using functions and defining the body of the function.

- `ReturnType` defines the type of the function return value. The type can one of the basic types described in section Section B.1, "Data Types".

- **AS** `ColumnName Type` defines the columns for this view and the data type for each column. Each column name must must be unique within the view and must follow the naming conventions specified later in this section within the stream. In other words, two columns within the same stream cannot have the same name. `Type` is one of the supported data types. Refer to Section B.1, "Data Types" for a list of supported types.

- **FROM** `TableName` | `ViewName` specifies specifies one or more input streams for this view. There may be any number of input streams defined for a Program View.

- **PROGRAM** is a collection of variables and methods that determines the output of this view. There may be any number of variables defined, but there has to be exactly one method defined for each input stream. Whenever a record is received on an input stream, the appropriate method is called to determine how the input will affect the view. Note that variables can be defined inside as well as outside of the methods. A variable defined outside the methods is form of "global" variable which is available to all the methods defined in this Program View.

- `VariableName` is the name of a variable, which must follow the naming convention standard specified in Section 1.2, "Names". A variable name must be unique within a program view.

- `VariableType` specifies the data type for the variable. The data type must be one of the types listed in Section B.1, "Data Types".

- `MethodName` is an optional name that can be specified for a method. If the MethodName is not provided, then the name is assigned to be the same as the `TableName`. The MethodName must follow the standard naming conventions specified in Section 1.2, "Names".

- `SplashProgram` is the actual SPLASH program that computes the output (if any). See Chapter 4, *SPLASH Programming Language* for information on how to write a SPLASH program.

2.6. Pattern View Definition

A Pattern View is an Aleri SQL construct that defines a Pattern Stream in the model. A Pattern View can be used to define complex relationships between records in one or more streams. It is a specialized form of a Materialized View and can be used anywhere a Materialized View can be used. The syntax to declare a Pattern View is as follows:

```
CREATE PATTERN VIEW ViewName
PRIMARY KEY ( KeyColumn [,...])
STORE StoreName [[
[| EXPIRES IN ExpiryDuration {SEC | MIN | HRS | HOUR | DAYS} [FROM ExpireFrom-
Column ]
SET ExpiryColumn [ NoOfTimes TIMES] ]
| [DECLARE VariableName [eventCache(CacheDefinition) | VariableType]]
] | [CREATE FUNCTION FunctionName {FunctionDefinition} RETURN ReturnType]
[...]]
AS
ColumnName Type [,...n]
PATTERN PatternDefinition [...]
```

where

- *ViewName* is the name of the Pattern View being defined. The name must be unique and must follow the naming conventions specified in Section 1.2, “Names”.
- **PRIMARY KEY** *KeyColumn* specifies one or more columns that will form the key for this view. A primary key must be defined for every Program View.
- **STORE** *StoreName* is the name of Store that will be used to hold retained data for this view. The persistence properties for the view are derived from the type of Store specified.
- **EXPIRES IN** specifies that a flag will be set on each record after the specified *ExpiryDuration* time period has elapsed. The column to be used for the expiry flag is specified by *ExpiryColumn* and the value in the column is incremented when the expiry time period has elapsed. The expired flag may be incremented more than once by specifying a value greater than one for the optional *NoOfTimes* parameter. By default, the expiry period is the time since the record was last updated (or created if it has never been updated); alternatively the expiry period can be the time since the timestamp contained in a column specified with *ExpireFromColumn.*
- *ExpiryDuration* specifies the time period after the expiry flag on a record is set. The time period is relative to the time the record was last updated unless an alternate time reference is specified in the *ExpiryFromColumn*. This must be a positive integer greater than 0.
- *ExpireFromColumn* is an optional parameter that specifies that the Expiry Duration is relative to the value in this column rather than the last time the row was updated or deleted. The specified column must be defined as part of the current stream and type date.
- *ExpiryColumn* specifies the name of the column in the current table that will be used to hold the expiry flag. This column's value will be incremented (updated) to indicate that the record has expired. This column must be of type integer.
- *NoOfTimes* is an optional parameter specifying the maximum number of times the expiry flag may be incremented. When the max is reached, the flag will no longer be incremented until it is reset to zero upon an update or new value in the ExpiryFromColumn. The default value is 1.
- **DECLARE** (optional) defines one or more variables and functions that are local to the view being defined. The variable can be one of the basic types or an eventCache. There may be any number of local variables and functions defined.
- *VariableName* is the name of the variable and must follow the standard naming conventions. Refer to Section 1.2, “Names” for more information. The variable names must be unique with the list of variables defined for the current view.

- `VariableType` is the data type of the variable being defined. The data type must be a supported data type or an eventCache. Refer to Section B.1, “Data Types” for a list of supported types.
- `CacheDefinition` is the definition for the event cache. Refer to Section 4.3.7, “Event Caches” for more information on defining and using even aches.

  Note:

  Although the syntax allows a user to specify more than one `INTERMEDIATE STORE` and `EXPIRES` clause for a derived stream, only the last is honored by the translator. The translator generates a warning when any clause is repeated.

- `FunctionName` is the name of the function being defined. The function name must be unique within the scope of the view being defined and must follow the standard naming conventions specified in the section Section 1.2, “Names”.
- `FunctionDefinition` is the body of the function and is defined using the SPLASH syntax. Refer to the section Section 4.5, “Functions” for more information on using functions and defining the body of the function.
- `ReturnType` defines the type of the function return value. The type can one of the basic types described in section Section B.1, “Data Types”.
- **AS** `ColumnName Type` defines the columns for this view and the data type for each column. Each column name must must be unique within the view and must follow the naming conventions specified later in this section within the stream. In other words, two columns within the same stream cannot have the same name. `Type` is one of the supported data types. Refer to Section B.1, “Data Types” for a list of supported types.
- **FROM** `TableName` | `ViewName` specifies specifies one or more input streams for this view. There may be any number of input streams defined for a Program View.
- **PATTERN** defines a pattern that needs to be detected and the action that needs to be performed when the pattern is detected. Each Pattern View must have at least one PATTERN clause.
- `PatternDefinition` is the definition for the pattern. Refer to the section Appendix C, *Pattern Matching Language* for information on defining Patterns.

2.7. Parameter Definition

A parameters can be used in an expression in place of a constant for a value that may need to be changed while the Sybase Aleri Streaming Platform is running. A parameter is synonymous to Global variables described in section Section 3.4, “Global”. The value of the parameter can be changed at runtime by issuing a command via the Command and Control interface or from any expression/rule from within the model.

The syntax for the parameter definition:

`DECLARE` ParameterName DataType[[DefaultValue]]
where:

- `ParameterName` is the name of a parameter. This name must be unique across all the objects in the SQL file.
- `DataType` is the data type for the parameter. It must be any of the supported data types.
- `DefaultValue` optionally specifies the default value for the parameter. If it's not specified, then it

defaults to NULL.

The defined parameters can be accessed in expressions via the `GETPARAM( )` function. The syntax for this function is as follows:

GETPARAM(ParameterName)

2.8. Global Function Definition

Just like Parameters, one or more global functions can also be declared. A Global function is available to any other functions (global or local) and rules/expressions that are defined after the function has been defined. The syntax for declaring global functions is as follows:

CREATE FUNCTION *`FunctionName`*{*`FunctionDefinition`*} RETURN *`ReturnType`*

where:

- *`FunctionName`* is the name of the function being defined. The function name must be unique within the scope of the model and must follow the standard naming conventions specified in the section Section 1.2, “Names”.
- *`FunctionDefinition`* is the body of the function and is defined using the SPLASH syntax. Refer to the section Section 4.5, “Functions” for more information on using functions and defining the body of the function.
- *`ReturnType`* defines the type of the function return value. The type can one of the basic types described in section Section B.1, “Data Types”.

2.9. Data Location Definition

The Data Location construct in Aleri SQL is used to define a corresponding Data Location object in AleriML. A Data Location defines the location and default characteristics of a source or target for the Sybase Aleri Streaming Platform. The Data Location definition goes hand in hand with Connector Definitions described below. The syntax for defining a Data Location is as follows:

`CREATE DATA LOCATION` *`LocationName`* TYPE IS *`TypeName`* [SET *`Property1=Value1[,Property2=Value2,...]`*]
where:

- *`LocationName`* is the name of the data location. This must be unique across the entire model and must follow the naming conventions described in section Section 1.2, “Names”.
- *`TypeName`* is the location type meaning the type of data that the connector provides. For example, xml_in, jdbc_in, or smtp_out.
- *`Property1`* is the first property or attribute that needs to be set for a given location type. The name of the property depends on the type of data location being specified. If the property name is a keyword or if it contains spaces then the it must be surrounded in double quotes. There may be any number of properties specified.
- *`Value1`* is the value for the property. If the value is a keyword or contains spaces it must be surrounded by single quotes.

Note:

The validity of the values supplied for `TypeName, Property1, Value1` not verified during compile time. If there are any errors it will be flagged during run time.

2.10. Connection Definition

Connections allow data to be imported from external sources into the Sybase Aleri Streaming Platform (inbound connectors) and export data from the Sybase Aleri Streaming Platform to external sources (outbound connectors). One or more connections can be associated with a view/table. Tables can be associated with both inbound and outbound connectors whereas views can be associated with only outbound connectors. The syntax for defining a connector is as follows

`CREATE` {input | output} CONNECTION *`ConnectionName`* FOR *`TableViewName`* {FROM *`LocationName`* | TYPE [IS] *`TypeName`*} [SET *`Property1=Value1[,Property2=Value2,...]`*]
where:

- *`ConnectionName`* is the name for the connection. It must be unique within the scope of a model and following the naming conventions described in the section Section 1.2, “Names”.
- *`LocationName`* is the name of a previously defined Data Location.
- *`TypeName`* represents the type of connector to create. This can only be provided if the *`LocationName`* parameter is not specified.
- *`Property1`* is the first property or attribute that needs to be set for a given location type. The name of the property depends on the type of data location being specified. If the property name is a keyword or if it contains spaces then the it must be surrounded in double quotes. There may be any number of properties specified.
- *`Value1`* is the value for the property. If the value is a keyword or contains spaces it must be surrounded by single quotes.

Note:

The validity of the values supplied for *`TypeName, Property1, Value1`* is not verified during compile time. If there are any errors it will be flagged during run time.

One or more Connections can be associated with a given Data Location.

The *`CREATE CONNECTION`* statement must follow the *`CREATE TABLE`* statement, or you get an error saying that the *`TableViewName`* does not exist.

2.11. Connection Group Definition

When one or more connections are defined in a model the question quickly arises, how to control the order in which the connectors are started up. This where the Connection Group comes in. One can assign a set of connectors to a group and specify whether the connectors should be started up or not. Also, if the connections in a group needs to be automatically started then the connections defined within a group are started in parallel. When multiple connection groups are defined each connection group that needs to be started is started in the order in which the groups are defined. For more information on Connection Groups refer to the document Section 2.13, “Access Control”.

The syntax for defining a Connection Group is as follows

`CREATE CONNECTION GROUP` *`GroupName`* (*`Connection1`* [,*`Connection2...`*]) TYPE [IS]

{start | nostart}

where:

- *`GroupName`* is the name for the connection group. It must be unique within the scope of a model and following the naming conventions described in the section Section 1.2, “Names”.
- *`Connection1`* is the name of a previously defined Connection that should be associated with this group. There must be at least one connection associated with every group.
- start | nostart specifies whether to automatically start the connections within the group when the Platform starts or not. If the connections are not automatically started they can be manually started via the command and control interface.

Note:

All connections specified within a group are started in parallel whereas the connection groups themselves are processed in the order in which they are defined.

A connection can be specified only once in only one group. If it appear more than once in a group or if it is specified in more than group it will be flagged as an error.

2.12. Distributed Model Definition

Aleri SQL allows a data model to be distributed across multiple servers through the definition of Modules and Clusters. Refer to the *Authoring Guide* for more information.

There are two parts to defining a distributed model. The first part defines a module. A module contains all of the Stores and Streams (tables and views) that will be executed on a single instance of the Sybase Aleri Streaming Platform.

A module is defined by enclosing a set of `CREATE (`*`store/table/view`*`)` statements within a **CREATE MODULE** statement. It uses the following Aleri SQL syntax:

`CREATE MODULE` ModuleName
`BEGIN`
ObjectDefinitions
`END` [;]
where:

- ***`ModuleName`*** is the name of the module. The module name follows the standard naming convention rules and must be unique across the entire model.
- **BEGIN** specifies the beginning of the module. All stores and streams defined after this point belong to the specified module.
- ***`ObjectDefinitions`*** represents any number of stores, streams, and derived streams that may be created. When created,these objects belong to the specified module.
- **END** signals the end of the module. This keyword must be provided before creating another module. If this is not provided, then all objects listed at the end of the file are considered to be part of the current module.

The second part to defining a distributed module involves assigning individual modules to run on specific servers. This is done via the following syntax:

`CREATE CLUSTER` ClusterName
ModuleName HostName CommandPort
[, ModuleName2 HostName2 CommandPort2...] [;]
where:

- *`ClusterName`* is the name of the cluster. It follows the standard object naming conventions and must be unique with a given model.
- *`ModuleName`* is the name of the module for which the instance information is to be specified. It may appear only once in each Cluster specification.
- *`HostName`* is the name of the host on which the Sybase Aleri Streaming Platform will be running.
- *`CommandPort`* is the TCP/IP port on which the Sybase Aleri Streaming Platform is listening for Command and Control information. This number must be positive and less than 65536.

More than one cluster section can be specified in a given model. The cluster to be used is decided at runtime. Every module may be reference by one cluster. Additionally, a cluster may be defined at any point in the model. But it's unnecessary to define all the modules before referencing one in a cluster definition.

```
CREATE CLUSTER
  cluster1 module1 "amazon.sybase.com" 11190,
           module2 "tigris.sybase.com" 11190

CREATE MODULE module1
BEGIN
  CREATE STORE store0 MEMSTORE;
  CREATE TABLE Stream1 FOR INSERT(
         a int64, PRIMARY KEY(a),
         STORE IS store0 );
END

CREATE MODULE module2
BEGIN
  CREATE STORE store1 MEMSTORE;
  CREATE MATERIALIZED VIEW Stream1Copy
         PRIMARY KEY(a)
         STORE IS store1
         AS SELECT * FROM Stream1;
  CREATE MATERIALIZED VIEW DerivedStream1
         PRIMARY KEY (a)
         STORE IS store0
         AS SELECT a.a, NULL*0 nullcol FROM Stream1Copy a;
END
```

2.13. Access Control

Access control can be specified either at the model level or for an individual stream. The syntax for specifying access control is:

GRANT [{*privilege*} [, ...]] [ON *StreamName*] TO [{*role*} [,...]]

where:

- *`privilege`* can have one of the following options: `control`, `query`, `subscribe`, `publish` or `all`. There may be more than one privilege given to each *`role`*. The default is `all`, which means that the specified *role(s)* has been granted all the possible privileges.

- `StreamName` is the name of the stream (table or view) for which the access control is being specified. If that is not provided, then this access control specification is put into effect at the model level. control.
- `role` is the name of the role for which the access control is being specified. There may be more than one role specified, in which case the specified privilege is applied to all of the specified roles. Note that the role does not have to exist at translation time.

There may be more than one `Grant` statement specified for a given stream, in which case the resultant Access Control for the stream is a union of all access controls specified. The Access Control setting for any intermediate streams (ones generated by the translator in order to create the specified Stream) defaults to the Access Control of the intermediate stream's inputs. This can only be modified by manually editing the generated AleriML file.

2.14. Aleri SQL Expressions

2.14.1. Join Expressions

A Join Expression follows the **FROM** keyword and is specified by including a `JoinType` followed by the elements of the Join. The syntax is:

```
FROM
{ TableName | ViewName } [AliasName]
JoinType { Table Name | ViewName } [AliasName ON]
[LeftExpression = RightExpression]
[AND LeftExpression...]
[JoinType...]
```

The supported Join types are `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN`.

A `Join Expression` allows data from two or more tables to be combined by matching records with common values in the specified fields. The join operation is performed before applying any complex filters (filters based on columns from more than one stream), aggregations, or computations. The expression following the `ON` keyword specifies how two tables are related to each other. This expression may contain more than one sub-expression concatenated by an `AND` clause.

Currently, each sub-expression is restricted to a simple between two columns, for example, `Alias1.Col1 = Alias2`.

The only currently supported operator is “=”.

One of the two columns must come from the first stream being joined, and the other column must come from the second stream being joined.

Note:

At least one side of a Join clause must be a key column.

As mentioned earlier, more than two tables can be Joined together. In this case, the Aleri SQL to AleriML translator will generate the AleriML to perform the Join in a single step as long as there is only one driving stream (that is, the many side of a one-to-many Join). If there are many driving tables in the `Join` expression, the translator will generate a sub-Join for every driving stream and will combine the results together.

Notes:

- The order of the Joins can be controlled to some extent by grouping the Joins using parentheses. See Section 2.14.1.2, “Examples of Joins” for details.
- The translator does not optimize based on the data in the tables because it does not know anything about them. This is the nature of streams and continuous queries: the query structures are defined before the data arrives. The translator only tries to optimize the generated AleriML to generate the least number of sub-Joins.

2.14.1.1. Types of Joins

As mentioned earlier, four different types of Joins are supported by the Sybase Aleri Streaming Platform. The type of join controls whether a record gets selected for processing. Here is an explanation of each of the Join types:

- **INNER JOIN** — When the columns being Joined do not match, no rows are selected from either of the tables being Joined.
- **LEFT JOIN** — When the columns being Joined do not match, all the rows belonging to the stream on the left side of the Join type are selected, and only those rows where the Join keys match are selected from the stream on the right side of the Join type. Usually the left side of the Join type in this case is the "many" side of a one-to-many relationship.
- **RIGHT JOIN** — This is the reverse of the LEFT JOIN. Note that any Join Expression can be written to be a RIGHT JOIN or a LEFT JOIN by rearranging the Join clause. The two Join types are supported only for convenience.
- **FULL JOIN** — A full Join occurs when the rows from both sides of a Join are included even if the Join keys do not match. Rules are generated to merge the columns being Joined (they contain values from both streams). If it is known that the two tables being joined do not have any overlapping keys with compatible row definitions, it is more efficient to do a union instead of a full outer Join.

An important point to understand in Joins is how the keys of the Join Stream are determined. This is especially true if a Derived Stream is defined with no aggregation. The reason for this is that the keys of the derived stream must match the keys of the intermediate Join Stream.

The rules for determining the keys of the Join Stream are:

- If a one-to-many relationship exists between the streams being Joined, the key of the stream on the “many” side becomes the key for the Join Stream and ultimately the key of the derived stream being defined via SQL.
- If a one-to-one relationship exists between the streams being Joined, the following rules apply for selecting a key for the Join table:
  - If the Join type is an INNER JOIN or a FULL JOIN, the key field(s) currently consists of the keys of the stream on the left side of the Join type.
  - If the Join type is a LEFT JOIN or a RIGHT JOIN, the key field(s) is derived from the stream on the left or right side of the Join type.

The Sybase Aleri Streaming Platform does not currently support many-to-many relationships.

If the key fields of the derived stream are required to be different from the keys of the final intermediate Join Stream, the stream must be defined as an Aggregate Stream using the `GROUP BY` clause.

2.14.1.2. Examples of Joins

A simple INNER JOIN — Joins `stream1` to `stream2`:

```
stream1 a INNER JOIN stream2 b ON (a.column1 = b.column1)
```

A more complex inner join first joins stream2 and stream3, and then joins the result to stream1:

```
stream1 a
  INNER JOIN (stream2 b INNER JOIN
              stream3 c ON b.column2 = c.column1)
  ON (a.column2 = b.column2)
```

A complex join using multiple join types — left joins `stream1` to the result of the right join between `stream2` and `stream3` and inner joins `stream4` to `stream1`:

```
stream1 a
  LEFT JOIN (stream2 b RIGHT JOIN
             stream3 c ON (b.column1 = c.column3) )
    ON (c.column2 = a.column1)
  INNER JOIN stream4 d ON (d.column4 = a.column1)
```

2.14.1.3. Restrictions on Joins

To be efficient and support the condition that all streams must have a primary key, the Sybase Aleri Streaming Platform places some restrictions on Joins to ensure that there are no duplicate key values in the output. The following rules and restrictions apply to Joins:

- All the keys of at least one side of the Join must be completely mapped on to fields on the other side of the Join. In other words, many-to-many relationships are not allowed.
- Both sides of a Join constraint must refer to columns in the tables that are being Joined. One of these columns must be from the table on the left side of the Join, and the other must be a column in the table on the right side of the Join.
- When defining a stream with no aggregation, the keys of the "many" side of a one-to-many or many-to-one relationship must be directly copied to the key fields of the stream being defined. In the case of a one-to-one relationship without aggregation, the keys of at least one of the streams must be directly copied to the key field of the view being defined. In a complex multi-level Join, these rules refer to the final Join being performed.
- When the view being defined has no aggregation, and it depends on a full-outer Join between two tables having a one-to-many relationship, the Join is effectively a left or right outer Join. This is because, for non-aggregated streams, the keys must be derived from the “many” side of a one-to-many relationship. If there is no matching row in the “many” side of a one-to-many relationship, not all the key columns of the “many” side will have a value, and they will not be eligible to be included in the target stream because key columns cannot be null.

2.14.2. Filter Expressions

A Filter Expression follows a `WHERE`, `HAVING` or `GROUP HAVING` keyword and has the following constraints:

- The result of the expression must be an integer value, where zero is the equivalent of `false` and any other value is treated as `true`.

- One or more expressions can be concatenated using AND/OR operators.
- The Filter Expression cannot contain aggregation functions.
- The Filter Expression in a `WHERE` clause cannot refer to columns in the stream being defined.
- The “*” notation cannot be used in a Filter Expression.

This example filters trades stream to get all the GOOGLE trades where the price >$400 or quantity > 1000:

```
CREATE STORE googleStore MEMSTORE;
CREATE MATERIALIZED VIEW GoogleTrades
  PRIMARY KEY(TradeId)
  STORE IS googleStore
  AS SELECT trd.TradeId, trd.Price, trd.Time, trd.Quantity
     FROM Trades trd
     WHERE trd.Symbol = "GOOG" AND
           (trd.Price > 400 OR trd.Quantity > 1000)
```

2.14.3. User Defined Function Library Declaration

Use the following commands to declare an external C library that contains one or more user defined functions.

`CREATE LIBRARY` *`LibraryName`* AS *`LibraryPath`* [;]
where:

- *`LibraryName`* is an alias of the library. This alias will be used within the Service Authoring Language to refer to the actual library.
- *`LibraryPath`* is the physical location and name of the C library file. Only an absolute path can be used.

Note:

The compiler does not check for the existence of this library file at compile time. It must exist at run time.

2.14.4. User Defined Function Declaration

Once the external library has been declared, any function within that library can be declared using the following syntax:

`CREATE FUNCTION` *`FunctionName`* (DataType [,...])
RETURN *`ReturnDataType`*
AS EXTERNAL LIBRARY *`LibraryName`* NAME *`ExternalFunctionName`*
LANGUAGE C [;]

where:

- *`FunctionName`* is an alias for the external function to be used within the Aleri SQL environment. The name must follow the standard naming conventions.

- `DataType` is the data type for the argument to the external function. It must be specified in terms of one of the data types that the Aleri SQL and the Sybase Aleri Streaming Platform understands. There can be any number of data types.
- `ReturnDataType` is the data type for the return value of the external function.
- `LibraryName` is an alias for the external library that is specified in the CREATE LIBRARY statement.
- `ExternalFunctionName` is the name of the external function in the specified external library.

  Note:

  The compiler does not check for the existence of this function during compile time.
- `LANGUAGE C` states that the external function is a 'C' library function. This currently has no effect but will be used in the future when Java® and possibly other languages are supported on the User Defined Function interface.

After this declaration, the function can be used in any rule as a built-in function.

2.15. Adding Comments in Aleri SQL

According to the SQL99 standard:

- multi-line comments begin with /* and end with */
- multi-line comments may be nested within a multi-line comment

Aleri SQL supports the multi-line comment syntax; it does not support any alternate syntax for single-line comments.

2.16. Current Restrictions on Aleri SQL Usage

The following restrictions apply to the current version of Aleri SQL but may be removed over time or as needed by customer requirements:

- When defining Joins, only individual columns can be used to Join tables. No other expression is currently allowed.
- When Joining tables, all keys of at least one side of the Join must be mapped to fields in the other side of the Join. Currently, many-to-many mappings are not supported.
- At least one side of a Join condition must refer to a key column. A Join condition involving two non-key columns is currently not supported.
- When defining Joins, "equals" is the only comparison operator currently supported.
- Self Joins are currently not supported. The equivalent can be achieved by creating view that is a "copy" and then joining to that.
- A stream may be used only once within a Join clause.
- When performing a UNION, the Sybase Aleri Streaming Platform expects all input records from all input streams to have unique key values. Otherwise, results may be unpredictable.

2.17. Best Practices When Writing a Data Model in Aleri SQL

- Use consistent and descriptive names when naming objects within the data model.
- Although the fields within a stream can be in any order, it is suggested that the key fields be listed first, followed by the attribute fields in alphabetical order.
- When a column in a derived stream has a complex rule and a second column in the same stream also needs to use the same complex rule, the second column should reference the first column instead of repeating the rule for the second time. Although this does not improve performance, it improves legibility and ease of maintenance.
- For performance reasons, avoid using INNER JOINs. Consider using an OUTER JOIN with a filter clause if required, that filters out rows with NULLs in the inner table's key fields. However, this currently applies only when the filter refers to columns from more than one table. If not, the filter will be applied before the Join, and the desired results are not achieved.
- The use of the same input stream multiple times in a Join clause is not possible in this version of the Sybase Aleri Streaming Platform. Do this by creating a copy of the table.
- Nested aggregations cannot be performed. If this is done, a runtime memory allocation error occurs and the Sybase Aleri Streaming Platform stops. This includes referring to another column within the stream being defined, which has an aggregation function.
- A name must be enclosed in double quotes if the name is the same as one of Aleri SQL keywords.
- Although using a ';' (semi-colon) for statement separator is optional, it is helpful to use this after every SQL statement. The reason is that the parser considers all the text up to a ';' (semi-colon) as a single statement and stops scanning the statement after it encounters the first error and moves to the next statement. As a result, if a '; '(semi-colon) is not used, any syntax errors will only be reported one at a time.

2.18. Example of an Aleri SQL Data Model

The following simple example shows a data model definition using Aleri SQL.

```
// Define a store that persists data
CREATE STORE Store1 LOGSTORE on '/tmp/sp/logstore' SYNC IS TRUE

// Define a store that does not persist data
CREATE STORE Store2 MEMSTORE

// Define a Static SourceStream Called Publishers
CREATE TABLE Publishers (
    publisher_id string, publisher_name string,
    state string, country string,
  PRIMARY KEY (publisher_id),
  STORE IS Store1,
  TYPE IS STATIC, RETAIN 10 MINUTES );

// Define a Static Stream Books, where data is retained by default
CREATE TABLE Books (
  isbn string, book_name string, author string,
  category string, publisher string, suggested_price double,
  PRIMARY KEY (isbn),
  STORE IS Store1,
  TYPE STATIC );

// Define a Dynamic Stream Called SalesData
```

```
CREATE TABLE SalesData (
    sale_id string, dealer string, isbn string, publisher_id string,
    sale_date date, quantity int32, unit_cost double,
  PRIMARY KEY(sale_id, dealer),
  STORE IS Store1,
  TYPE DYNAMIC, RETAIN 30 DAYS );

// Define a Materialized View called TitleSalesByDealer
// for all Title and Dealer combinations where the dealer has sold
// more than 1000 copies of a book published by ACME Publishing.
CREATE MATERIALIZED VIEW TitleSalesByDealer
  PRIMARY KEY (isbn, dealer)
  STORE Store2
  AS SELECT a.dealer AS dealer, b.isbn,
      MIN(b.book_name) AS title_name,
      MIN(c.publisher_name) AS publisher_name,
      SUM(a.quantity) AS total_quantity,
      SUM(a.quantity * a.unit_cost) AS total_sales,
      CASE WHEN (SUM(a.quantity * b.suggested_price)-total_sales)>0
           THEN SUM(a.quantity * b.suggested_price)-total_sales
           ELSE 0.0
      END AS total_discount
     FROM SalesData a
          LEFT JOIN Books b ON a.isbn = b.isbn
          LEFT JOIN Publishers c ON a.publisher_id = c.publisher_id
     WHERE c.publisher_name = 'ACME Publishing'
     GROUP BY a.dealer, b.isbn
     HAVING total_quantity > 1000
```

Note:

The reference to `total_sales` in the rule for `total_discount` does not refer to an AliasName. This tells the Sybase Aleri Streaming Platform to substitute the rule used for the `total_sales` column in place of this reference. This kind of reference has no impact on efficiency but improves maintainability of the code.

2.19. Running a model written in Aleri SQL

In order to run a data model written in Aleri SQL, the model must first be "compiled" into AleriML. The Sybase Aleri Streaming Platform can then read the AleriML file and run the model. The Sybase Aleri Streaming Platform includes the **sp_sql2xml** tool (also referred as the translator) which can be used to translate a data model written in Aleri SQL into an AleriML file that can be run on the Sybase Aleri Streaming Platform. Refer to the Utilities Guide for more information.

Chapter 3. Authoring in AleriML

Both the Aleri Studio and Aleri SQL produce a data model in AleriML. AleriML files are loaded directly into the Sybase Aleri Streaming Platform for execution.

You can create models directly in AleriML as an alternative to using the Studio or Aleri SQL. While there's more of a learning curve, experienced users may find it more efficient to create or edit data models. Data models created in AleriML can still be loaded into the Aleri Studio in order to visualize, check for violations, and test the model.

AleriML is standard XML that conforms to the AleriML Schema. This chapter describes the elements and attributes within that schema, and gives examples of AleriML.

3.1. XML Preliminaries

AleriML documents should begin with the following header that describes the version of XML and the character set:

```
<?xml version="1.0" encoding="UTF-8"?>
```

As with all XML documents, comments in AleriML begin with "<!--" and end with "-->". They may extend over a line. Comments cannot be nested.

All “id” attributes must be globally unique within a data model

All AleriML elements have optional “name” and “documentation” attributes. These attributes, which give you the ability to annotate the data model with comments, won't be listed below.

3.2. Platform

An AleriML file must have a single Platform element encapsulating all other elements.

```
<Platform
version="version string"
[moneyPrecision="number of decimal digits in money datatype"]
[restrictAccess="access restriction annotations"]>
[StartUp]
[Global]
{Cluster | Module | DataLocation | Store | Stream}*
</Platform>
```

Within the Sybase Aleri Streaming Platform element, there is an optional StartUp element and an optional Global element. These are followed by any number of Cluster, Module, DataLocation, Store, and Stream elements in any order. The following sections describe each of these elements.

Attributes:

version	This string should be “3.0”. Data models for older versions of the Sybase Aleri Streaming Platform might have different version attributes.
moneyPrecision	This attribute specifies the number of decimal digits (digits to the right of the decimal point) in the “money” datatype. The default is 4.
restrictAccess	This attribute allows you to restrict certain types of access for specific users. It may be specified at the Sybase Aleri Streaming Platform model level or stream level. The model-level attribute provides control over general access and defaults

for all streams. If a stream has its own restrictAccess attribute, it completely determines access to that stream. Otherwise the model-level attribute is used. If the restrictAccess attribute is missing at the model level, all users would have access.

Since the metadata streams don't have an explicit restrictAccess attribute, subscription access is always determined by the model-level attribute. For more information on metadata streams see Appendix E, *Aleri Metadata Streams*

This attribute contains a sequence of colon-separated pairs, with each pair separated by a space. For example:

```
wheel:control development:control
```

The first element of each pair is the role, which describes a set of users. The role is a translation of a user group in the operating system, as it gets read through the PAM interface. See the *Administrator's Guide* for more details on roles.

The second element of each pair specifies the type of access granted, including “control”, “query”, “subscribe”, “publish” or “connect”.

- *`control`* lets you execute the control commands that effect the entire model, such as stopping or debugging. It only impacts the model level.
- *`query`* lets you execute queries and updates through the SQL interface (including the finalizer statements).
- *`subscribe`* lets you to subscribe to the content of the stream.
- *`publish`* lets you publish data to the stream. It is only applicable to source streams.
- *`connect`* lets you connect to the model and receive the basic schema information, such as the names of the streams and fields. Without this permission, you won't be allowed to log in. *`connect`* only impacts the model level. Any role given any access at that level also implicitly receives *`connect`* access.

The previous example means that members of the role “wheel” or “development” can issue control commands, such as stopping the model by the **sp_cli** utility.

Here is an example of all access controlled at the model level:

```
<Platform restrictAccess="gcontrol:control gsub:subscribe gq:query
gpub:publish">
<Store file="store" id="store"/>
<SourceStream id="filterInput" store="store">
  ...
</SourceStream>

<FilterStream id="filter" istream="filterInput"
ofile="output/filter.out" store="store">
  ...
</FilterStream>
</Platform>
```

An example of all stream access controlled at the stream level:

```
<Platform    restrictAccess="gcontrol:control gsub:connect
gq:connect
gpub:connect">
<Store file="store" id="store"/>
<SourceStream id="filterInput" store="store"
    restrictAccess="gpub:publish"
>
  ...
</SourceStream>

<FilterStream id="filter" istream="filterInput"
ofile="output/filter.out" store="store"
    restrictAccess="gsub:subscribe gq:query"
>
  ...
</FilterStream>
</Platform>
```

In the previous example, roles that can access the streams are also listed at the model level with "connect" access. Otherwise, a user might not be able to log into the model since their role was restricted only to publish and subscribe. Another approach is to have a separate role for connection and grant both the connection and subscription roles to a user who needs to subscribe.

```
<Platform     restrictAccess="gcontrol:control gconnect:connect">
<Store file="store" id="store">
<SourceStream id="filterInput" store="store"
    restrictAccess="gpub:publish"
<
  ...

</SourceStream>

<FilterStream id="filter" istream="filterInput"
ofile="output/filter.out" store="store"
    restrictAccess="gsub:subscribe gq:query"
>
  ...

</FilterStream>
</Platform>
```

3.3. StartUp

The StartUp element is an optional, and advanced, component of a data model. It describes the order in which connections start when the data model is loaded by the Sybase Aleri Streaming Platform.

A Startup element contains any number of ConnectionGroups:

```
<StartUp>
[ConnectionGroup]*
</StartUp>
```

Each ConnectionGroup is a collection of references to specific connections:

```
<ConnectionGroup
[type="{start|nostart}"]
[id="name of connection group"]>
[<ConnectionRef connection="name of connection"/>]*
</ConnectionGroup>
```

The ConnectionGroups within a StartUp element are started either automatically or manually, depending on the type. The automatic ones start in order: the connections in the first ConnectionGroup are started first, the connections in the second are started second, and so forth.

Attributes:

type	This attribute specifies whether to start the connections in the ConnectionGroup automatically (the option `start`) or later manually (the option `nostart`). The default is `start`.
id	This is the name of the ConnectionGroup. It is used when issuing a command (via the **sp_cli** utility) to start the Connections in the ConnectionGroup. There is no default value.

3.4. Global

The Global element encapsulates a set of variables and functions. Streams can read or write these variables, and can use these functions. Global variables can also be altered at runtime via the Command and Control interface.

The Global element is a simple block of text, with variable and function definitions written in the SPLASH scripting language.

```
<Global>
...
</Global>
```

See Chapter 4, *SPLASH Programming Language* for more information about variable and function declarations. For example, here is a block that defines a global variable `depth_of_book` (initially 10) and a function `change_currency`:

```
<Global>
int32 depth_of_book := 10;
double change_currency(double val) { return val * 1.57; }
</Global>
```

Global variables can be modified outside of a running instance, using the Command and Control interface or directly through the **sp_cnc** program.

Note:

Changing the value of a global variable does not cause the model to regenerate the data based on the new value. For instance, if a Filter Stream filters rows based on the value of a global variable, changing the value changes the filtering of the new rows but won't affect the old rows. To regenerate data after changing the value of a variable, use the Dynamic Sybase Aleri Streaming Platform Modifications, as described in the *Administrator's Guide*.

3.5. Cluster

A Cluster is a description of machines and port numbers for running the Sybase Aleri Streaming Platform across a number of machines. It has the syntax

```
<Cluster
id="name of cluster">
(Node)*
</Cluster>
```

where each node has the syntax

```
<Node
[module="name of module"]
machine="name of machine"
```

```
commandport="port number" />
```

See the *Administrator's Guide* for more information about setting up a cluster to run a data model.

3.6. Module

A Module is a collection of other modules, data locations, stores, and streams used to encapsulate a portion of the data model for use in a clustered setting. It has the syntax

```
<Module
id="name of module" >
{Module | DataLocation | Store | Stream}*
</Module>
```

See the *Administrator's Guide* for more information about setting up a cluster to run a data model.

3.7. DataLocation

A DataLocation is a description of an external data source, a place where the data model will automatically load data, or a sink, a place where the data model will automatically send data. It has the syntax

```
<DataLocation
id="name of data location"
type="type of data location">
[<LocationParam name="parameter name" value="parameter value"/>]*
</DataLocation>
```

Each LocationParam is simply a name-value pair for a parameter.

Attributes:

id	This is the name of the data location, which is used in InConnections and OutConnections (see below).
type	This is the type of the data location. There are a number of different types, for example, `jdbc_in`, `xml_out`.

Here are a few example data locations:

```
<DataLocation id="xml_file_input" type="xml_in">
<LocationParam name="dir" value="/tmp"/>
</DataLocation>
<DataLocation id="sqlsrv_output" type="db_out">
<LocationParam name="dbtype" value="mssql"/>
<LocationParam name="server" value="vilcanota.sybase.com"/>
<LocationParam name="port" value="1433"/>
<LocationParam name="user" value="sa"/>
<LocationParam name="password" value="tiger"/>
<LocationParam name="database" value="aleri"/>
</DataLocation>
```

See the *Authoring Guide* for more information about data locations.

3.8. Store

A store is a repository for the records of one or more streams. There are three kinds of stores: Stateless Stores, Memory Stores, and Log Stores.

3.8.1. Stateless Store

To define a Stateless Store, which keeps no records, use

```
<Store id="name of store" kind="stateless"/>
```

Attributes:

id	This is the name of the store, which is used in stream definitions.

Stateless Stores are permitted only for insert-only streams and FlexStreams; see Section 3.9.1, “Insert-only Streams” for more detail.

3.8.2. Memory Store

To define a Memory Store, which keeps its records in memory, use

```
<Store id="name of store" kind="memory"
[index="{tree|hash|list}"]
[indexSizeHint="initial size of hash table, in units of 1024"]/>
```

Attributes:

id
: This is the name of the store, which is used in stream definitions.

index
: This determines the data structure used for indexing records. The default value is “tree”.

- Use `tree` for binary trees. Binary trees are predictable in use of memory and consistent in speed.
- Use `hash` for hash tables. In many situations, hash tables are faster, but they often consume more memory.
- Use `list` for lists. This keeps the records in order of insertion, and thus can be useful in looping through the records. You should use this option only if you are certain there will be few deletes in the streams kept in this store. Inserts and updates of records are efficient, but deletes leave space that is not reclaimed.

indexSizeHint
: This optional attribute determines the initial number of elements in the hash table, when using `hash`. The value is in units of 1024. Setting this higher consumes more memory, but reduces the chances of spikes in latency.

3.8.3. Log Store

To define a Log Store, which keeps records stored on disk for recovery, use

```
<Store id="name of store" kind="log"
file="directory name"
[sync="{true|false}"]
[fullsize="maximum size in megabytes"]
[sweepamount="sweep size in megabytes"]
[reservePct="reserve size in percent"]
[ckcount="record count"]/>
```

Attributes:

id	This is the name of the store, which is used in stream definitions.
file	The file name specifies the name of a directory into which the persisted store will be written.
sync	This attribute determines how frequently the store commits records to disk. The default is `false`. When sync is `false`, records are committed at periodically. In the event of a failure, data received since the last commit will be lost. When sync is `true`, records are committed immediately upon receipt. Setting this attribute to `true` makes the Platform run more slowly.
fullsize	This attribute specifies the maximum size of the Log Store, in megabytes. The default is 8 megabytes.
sweepamount	This indicates the amount of data, specified in megabytes, that is examined when trying to reclaim unused space. The default value is 20% of the fullsize.
reservePct	This indicates the percentage of the log store size to keep as the free space reserve. The default value is 20 percent.
ckcount	The "checkpointing count" attribute lets you establish the maximum number of records written between intermediate metadata. The default is 10,000.

3.9. Stream

A stream is a processing node in the data model. There are ten stream forms: Source, Copy, Filter, Union, Compute, Extend, Aggregate, Join, Flex, and Pattern.

3.9.1. Insert-only Streams

Some streams are considered to be “insert-only,” meaning that they process inserts but not updates or deletes. The following streams are insert-only:

- An insert-only Source Stream.
- A Union Stream, Compute Stream or Filter Stream whose inputs are insert-only.
- A Join Stream whose “many” inputs in the many-to-one Joins are insert-only, and whose “one” inputs are “static” streams.

3.9.2. Common Attributes & Elements

Each stream type has a slightly different set of attributes and elements. Nevertheless, there are some common features of all stream.

```
<StreamType
id="name of stream"
store="name of Store"
[ofile="output file name"]
[type="{static|dynamic}"]
[expiryTime="wait time in seconds"]
[expiryField="name of int32 column to increment after expiryTime"]
[expiryTimeField="name of date column for calculating expiry"]
[expiryMaxValue="number of times to increment expiryField"]
[restrictAccess="access restriction annotations"]
[type="{static|dynamic}"]
[oldid="old name of stream"]
```

```
[convdst="destination stream for conversion"] >
...
</StreamType>
```

Attributes:

id	This attribute specifies the name of the stream. It must be unique within the data model.
store	This attribute specifies the store that will be used to hold the records of the stream.
ofile	This optional attribute specifies the name of an output file to which all records will be written if the Sybase Aleri Streaming Platform shuts down cleanly. The records are written in XML format. This attribute is used mainly in debugging. No output file is created if the ofile attribute is missing.
type	This optional attribute gives hints to the Sybase Aleri Streaming Platform for optimization. It may be either `static`, denoting that data is loaded once when the Sybase Aleri Streaming Platform starts, or `dynamic`, denoting that data changes frequently. The default is `dynamic`.
expiryTime	The expiryTime attribute specifies the age, in seconds, that must expire before a record's expiryField is incremented.
expiryField	specifies the name of an int32 column that will be set to 0 when a record is modified, and incremented (up to *`expiryMaxValue`* when a record ages to *`expiryTime`* seconds.
expiryTimeField	specifies the optional name of a date column that will be used to calculate the expiryTime. If this attribute is omitted, expiryTime will be computed from the time the record was last modified.
expiryMaxValue	is the number of times the expiryField column will be incremented. When the expiryField reaches this value, the record is no longer updated until an update comes from the outside. The default is 1.
restrictAccess	is used to limit access to the stream. See Section 3.2, “Platform” for more information about restrictAccess.
oldid	is used during the dynamic model modifications to rename the stream from oldid name to the id name. See the *Administrator's Guide* for details. The attribute has no effect otherwise.
convdst	is used in conversion models that convert the contents of the Source Streams of the main model during a dynamic model modification. See the *Administrator's Guide* for details. This attribute is not allowed in the normal models.

3.9.2.1. Column

The Column element describes the name and datatype of a column. The syntax is

```
<Column
name="name of column"
[key="{true|false}"]
[autogen="{true|false}"]
[datatype="{int32|int64|money|double|date|timestamp|string}"]>
```

Attributes:

name	This is the name of the column.
key	This attribute should be set to `true` if the column is one of the primary key columns in the stream. By default, the attribute is `false`.
autogen	This attribute should be `true` if the data in the column should be generated automatically by the stream. By default, the attribute is `false`. (This attribute works only for Columns within Source Streams.)
datatype	This attribute specifies the type of the data in the column. The default value is int32.

Column elements appear in Source, Flex, and Pattern Streams.

3.9.2.2. ColumnExpression

The ColumnExpression element describes the name of a column and an expression for computing it. It has the syntax

```
<ColumnExpression
name="name of column"
[key="{true|false}"]>
...
</ColumnExpression>
```

Attributes:

name	This is the name of the column.
key	This attribute should be set to `true` if the column is one of the primary key columns in the stream. By default, the attribute is `false`.

An expression must appear between `<ColumnExpression...>` and `</ColumnExpression>`. Expressions follow the syntax described in Section 1.3, “Expressions”.

ColumnExpression elements appear in Compute, Extend, Aggregate, and Join Streams.

3.9.2.3. FilterExpression

```
<FilterExpression>
...
</FilterExpression>
```

An expression of type int32 must appear between `<FilterExpression>` and `</FilterExpression>`. Expressions follow the syntax described in Section 1.3, “Expressions”.

FilterExpression elements appear optionally in Source Streams, and in Filter Streams.

3.9.2.4. InConnection and OutConnection

InConnection and OutConnection elements describe an external source of data. The syntax is

```
<InConnection
```

```
name="name of connection"
location="name of location" >
[<ConnectionParam name="parameter name" value="parameter value"]*
</InConnection>

<OutConnection
name="name of connection"
location="name of location">
[<ConnectionParam name="parameter name" value="parameter value"]*
</OutConnection>
```

Attributes:

name	This is the name of the connection that can be used in StartUp elements. See Section 3.3, “StartUp” for more information.
location	This attribute specifies the name of a DataLocation element. See Section 3.7, “DataLocation” for more information.

Within the InConnection and OutConnection elements are a number of ConnectionParam elements. These override or add to the LocationParam elements in the corresponding DataLocation.

InConnection elements appear optionally in Source Streams. OutConnection elements appear optionally in any stream.

3.9.2.5. Local

The Local element encapsulates a set of variables and functions for a particular stream. It behaves just like the Global block (see Section 3.4, “Global”), except that only the stream that contains the Local element can read or write the variables or use the functions. The syntax is

```
<Local>
...
</Local>
```

Local elements appear optionally within Compute, Extend, Aggregate, Join, Flex, and Pattern Streams.

3.9.2.6. InputWindow

An Input Window limits the view of an input stream, based on the age of records or the number of records. For instance, you can set an input window so that a stream sees only the last 10 records of one of its input streams. The stream is kept consistent with that view by forcing deletes through the stream, as if those deletes came from the input stream.

The syntax is

```
<InputWindow
[stream="name of stream"]
[type="{records|time}"]
[value="{number of records|seconds}"]
[slack="number of records"]/>
```

Attributes:

stream	This attribute specifies the input stream on which the window is used.

type	This attribute specifies the policy. Use `time` to limit the view of the input stream based on a number of seconds, or the `records` to limit the view of the stream to a number of records.
value	This attribute describes the amount of time or number of records to hold. It must be an integer greater than 0. If type is set to `time`, a value of 100 means that all records older than 100 seconds are deleted from the view. If type is set to `records`, a value of 100 means that all but the last 100 records are deleted from the view. The default is 1.
slack	This attribute is meaningful only when type is `records`, and is used to control the purging of data. For example, if `value` is 1000 records and `slack` is set to 500, the oldest 500 records will be purged when the stream grows to 1500 records. Therefore, at any point in time, the size of the table will be between 1000 and 1500 rows. The default is 1.

InputWindow elements appear optionally in Source, Copy, Union, Filter, Compute, Extend, and Aggregate Streams.

3.9.3. Source Stream

A Source Stream is a stream whose input comes from the external world. Incoming messages can be applied to a Source Stream as inserts, updates, deletes, or upserts.

The syntax for a Source Stream is as follows:

```
<SourceStream ...
[insertOnly="{true|false}"]
[convsrc="name of original stream"] >
{InputWindow | OutConnection}*
(Column)+
[FilterExpression]
[InConnection]*
<SourceStream>
```

Attributes:

insertOnly	When set to `true`, this attribute specifies that the Source Stream ignore all upserts, updates, and deletes. It allows further optimizations to be made, allows the stream to be put in a Stateless Store, and enables certain joins. The default is `false`.
convsrc	This attribute is used during dynamic model changes to convert the contents of Source Streams. See the *Administrator's Guide* for details. This attribute is not allowed in the normal models.

Examples:

```
<SourceStream id="Currencies" store="store"/>
<Column key="true" name="Currency" datatype="string"/>
<Column key="false" name="Country" datatype="string"/>
</SourceStream>
```

This defines a Source Stream named `Currencies` whose records are stored in `store`. Each record contains a Currency and a Country field. The Currency field is the primary key.

```
<SourceStream id="Currencies" store="store" >
<Column key="true" name="Currency" datatype="string"/>
<Column key="false" name="Country" datatype="string"/>
<InputWindow type="records" value="1000" slack="500"/>
```

```
</SourceStream>
```

This defines a Source Stream that retains between 1000 and 1500 records, and will purge the stream down to 1000 records when it exceeds 1500 records.

3.9.4. Copy Stream

A Copy Stream is a stream whose input comes from exactly one stream. It holds records from the input stream. A Copy Stream is usually used in conjunction with an InputWindow to hold a view of a stream.

The syntax for a Copy Stream is as follows:

```
<CopyStream ... istream="name of input stream" >
{InputWindow | OutConnection}*
</CopyStream>
```

Attributes:

istream	This attribute specifies the name of the input stream.

Example:

```
<CopyStream id="CurrenciesSubset" store="store" istream="Currencies">
<InputWindow stream="Currencies" type="records"
       value="1000" slack="500"/>
</CopyStream>
```

The stream CurrenciesSubset copies data from Currencies. The InputWindow will truncate the stream to 1000 records when it exceeds 1500 records.

3.9.5. Union Stream

A Union Stream is a stream whose input comes from one or more other streams, and whose output is a set of records representing the union of the inputs.

The syntax for a Union Stream is as follows:

```
<UnionStream ... istream="name of input streams separated by spaces"
[mergeKeys="{true|false}"]>
{InputWindow | OutConnection}*
</UnionStream>
```

Attributes:

istream	This attribute specifies the name of the input streams.
mergeKeys	This attribute allows the Union Stream to handle inserts or deletes for the same keys from different inputs. For instance, suppose the stream receives an insert for a key from input stream Input1, and another insert for the same key from input stream Input2. When the attribute is set to `false` (the default), the second insert will be rejected. When the attribute is set to `true`, the second insert will be turned into an update. Similarly, deletes on the same key from different streams will not cause errors when the attribute is set to `true`.

Example:

```
<UnionStream id="AllCurrencies" store="store"
     istream="Currencies OldCurrencies"/>
```

This specifies a stream that is the union of “Currencies” and “OldCurrencies”.

Note:

Unpredictable runtime results may occur a row from one input stream has the same key as a row from another input stream.

3.9.6. Filter Stream

A Filter Stream is a stream whose input comes from exactly one stream, and the output consists of a subset of the records from the input stream. qna Each input record is evaluated against one or more filter expressions. If one or more of the filter expressions evaluates to 0, the record does not become part of the Filter Stream.

The syntax for a Filter Stream is as follows:

```
<FilterStream ... istream="name of input stream" >
{InputWindow | OutConnection}*
FilterExpression
</FilterStream>
```

Attributes:

istream	This attribute specifies the name of the input streams.

```
<FilterStream id="CurrentCurrencies" store="store"
      istream="Currencies">
<FilterExpression>
Currencies.CurrentTime = undate('2005-08-10 09:58:00')
</FilterExpression>
</FilterStream>
```

This defines a Filter Stream called `CurrentCurrencies` whose records are stored in the store named `store`. The filterExpression specifies that only those records whose CurrentTime value is equal to the time `2005-08-10 09:58:00` will be passed into the Filter Stream. More information on the expression language can be found below.

3.9.7. Compute Stream

A Compute Stream is a stream whose input comes from exactly one stream. Its output consists of a new set of records whose fields are computed from the fields in the input.

The syntax for a Compute Stream is as follows:

```
<ComputeStream ...
istream="name of input stream"
[permitKeyChange="{true|false}"]>
{InputWindow | OutConnection}*
[Local]
(ColumnExpression)+
</ComputeStream>
```

Attributes:

istream	This attribute specifies the name of the input stream.
permitKeyChange	This advanced attribute allows the Compute Stream to change the primary key structure. When this attribute is `false` (the default), the primary key columns of the input stream must be copied, without further computation, into the Compute Stream (primary key columns can be added, however, in the Compute Stream). When this attribute is `true`, the primary key columns can be changed.

Note:

Setting the attribute permitKeyChange to `true` can result in errors at run time. For instance, if there is an event to update a row, the computation might try to update a row that does not exist.

Example:

```
<ComputeStream id="NormalizedCurrencies" store="log"
       istream="CurrentCurrencies">
<ColumnExpression key="true" name="Currency" >
CurrentCurrencies.Currency
</ColumnExpression>
<ColumnExpression name="Location" >
CurrentCurrencies.LocationRule
</ColumnExpression>
<ColumnExpression name="ExchRate" >
CurrentCurrencies.ExchRate *  100.0
</ColumnExpression>
</ComputeStream>
```

This defines a Compute Stream called `NormalizedCurrencies` whose records are stored in the Store named "store". The key field is `Currency`, and the expression for computing the Currency field — the `CurrencyRule` rule — just passes it along unchanged. The only non-trivial computation done on the fields is in the ExchRate field, whose value is multiplied by 100.0.

3.9.8. Extend Stream

A Extend Stream is a stream whose input comes from exactly one stream. Its output consists of a new set of records whose fields are computed from the fields in the input. The columns are those from the input stream, plus some new columns

The syntax for an Extend Stream is as follows:

```
<ExtendStream ... istream="name of input stream">
{InputWindow | OutConnection}*
[Local]
(ColumnExpression)+
</ExtendStream>
```

Attributes:

istream	This attribute specifies the name of the input stream.

Extend Streams provide a bit of reuse. If, for instance, you add a column to the input stream of an Extend Stream, the new column will be automatically carried along. You can also use an Extend Stream to override the computation of columns. For example, if you want to change the computation of a column, you can simply specify a new ColumnExpression for that column name.

Example:

```
<ExtendStream id="NewCurrencies" store="log"
      istream="NormalizedCurrencies">
<ColumnExpression name="ExchRate" >
NormalizedCurrencies.ExchRate *  109.0
</ColumnExpression>
<ColumnExpression name="AnotherRate" >
NormalizedCurrencies.ExchRate *  200.0
</ColumnExpression>
</ExtendStream>
```

This stream extends the `NewCurrencies` stream. It computes the `ExchRate` column using a different expression, and adds the `AnotherRate` column.

3.9.9. Aggregate Stream

An Aggregate Stream is a stream whose input comes from exactly one other stream, source or derived. Its output contains a new set of records whose fields are computed from the records in the input. Records in the input stream are grouped according to common values as specified in the Group expression. The output of the Aggregate Stream contains a single record for each Group, and this record can contain values that are computed across all members of the group. Thus, the number of records in an Aggregate Stream is less than or equal to the number of records in the input.

The syntax for an Aggregate Stream is as follows:

```
<AggregateStream  ... istream="name of input stream" >
{InputWindow | OutConnection}*
[Local]
(ColumnExpression)+
[<GroupOrder [ascend="{true|false}"]>...</GroupOrder>]*
[<GroupFilter>...</GroupFilter>]*
</AggregateStream>
```

Components:

istream	This attribute specifies the name of the input stream.
GroupOrder	These optional elements specify a lexicographic order on the records of each group. GroupOrder elements are often used in conjunction with the `first`, `last` and `rank` functions; refer to Appendix B, *Data Types, Operators and Functions* for more information about these functions. Each GroupOrder element contains expression, and may include an ascend flag. If the flag is not specified its value defaults to `true`.
GroupFilter	These optional elements specify a filter that is applied before the group is collapsed into a single row. The rule that is referenced by a GroupFilter element must return a value of type `int32` as with FilterExpression elements. A record in the group passes the filter if the expression evaluates to a value other than 0.

Example:

```
<AggregateStream id="MaxRateCurrencies" store="store"
         istream="NormalizedCurrencies">
```

```
<ColumnExpression key="true" name="Location">
NormalizedCurrencies.Location
</ColumnExpression>
<ColumnExpression key="false" name="ExchRate">
max(NormalizedCurrencies.ExchRate)
</ColumnExpression>
</AggregateStream>
```

This defines an Aggregate Stream called `MaxRateCurrencies` whose records are stored in `store`. The rows from the input table are grouped according to each one's value in the Location column. The output row consists of this Location value and the maximum ExchRate value from the group.

3.9.10. Join Stream

A Join Stream matches records from two or more input streams to produce records in a single output stream. Records are "matched" when they have matching values in one or more columns.

The syntax for a Join Stream is as follows:

```
<JoinStream ... istream="name of input streams separated by spaces" >
{InputWindow | OutConnection}*
(Join)+
[Local]
(ColumnExpression)+
</JoinStream>
```

Attributes:

istream	This attribute specifies the name of the input streams.

The Join elements specify how to match the records of the input streams. Join elements have the syntax

```
<Join
table1="name of stream"
table2="name of stream"
[type="{leftouter|fullouter|inner}"]
constraints="equality constraints"
[secondary="{true|false}"]
[optimize="{true|false}"]/>
```

Attributes of the Join elements:

table1	This attribute specifies the name of the first stream in the join. This stream is the "left" stream in the case of a `leftouter` join.
table2	This attribute specifies the name of the second stream in the join. This stream is the "right" stream in the case of a `leftouter` join.
type	This attribute specifies the type of join. In a `leftouter` join, if a row in the first stream has no match in the second stream, the second stream's values are assumed to be null. In a `fullouter` join, if a row in either stream has no match in the other, the other's values are assumed to be null. In a `inner` join, the rows in each stream must match. The default is `leftouter`.
constraints	This attribute specifies how the rows match. It must be sequence of equality con-

	straints among columns, with each equality is separated by a space). The left side of each equality must be a column from the stream mentioned in `table1`; the right side must be a column from `table2`.
secondary	This attribute specifies whether to build a secondary index for the first stream. When set to `true`, an auxiliary data structure is built. This speeds up computations when the second stream changes frequently, but consumes more memory. The default is `false`.
optimize	This attribute is reserved for future use.

Each Join must be one-to-one or many-to-one. To enforce this condition, the ColumnExpressions for the key columns must satisfy two constraints:

- The expressions may refer only to key columns from the input streams, and they must either be copies of the key columns or combinations of key columns using the `firstnonnull` function (see below).
- The key columns of at least one of the input streams must be used in the ColumnExpressions for the keys of the Join Stream. For example, if there are two input streams `S` and `T`, where `S` has key columns `k1` and `k2`, and `T` has key columns `m1` and `m2`, the ColumnExpressions for the keys of the Join must include `k1` and `k2` or `m1` and `m2`. It would be illegal to use only `k1` and `m1`.

Note:

The type may be `inner` only when all the input streams to the Join are “insert-only”; the definition is given above in Section 3.9.1, “Insert-only Streams”. These restrictions maintain the correctness of joins under updates and deletes while maintaining efficiency. The restrictions may be removed in a future release.

Example:

```
<JoinStream id="CountryCurrencies" store="store"
istream="Currencies Locations">
<Join type="leftouter" table1="Currency" table2="Locations"
constraints="Location=Location"/>
<ColumnExpression key="true" name="Currency">
Currencies.Currency
</ColumnExpression>
<ColumnExpression name="Location">
Currencies.Location
</ColumnExpression>
<ColumnExpression name="Country">
Location.Country
</ColumnExpression>
</JoinStream>
```

This stream matches the records in the `Currencies` stream with records in the `Locations` stream, where the records have the same value `Location` column. Because it is a `leftouter` join, if there is no matching row in `Location`, the value of `Location.Country` is null.

In other words, this stream shows how to get the country for a currency.

3.9.11. FlexStream

A FlexStream is a programmable stream whose input comes from one or more other streams, and whose output is generated by one or more small programs written in a special programming language called

SPLASH (Streaming Platform LAnguage SHell). Refer to Chapter 4, *SPLASH Programming Language* for information about the programming language.

The FlexStream allows you to build logic that goes beyond the usual relational operations. It has, for instance, a concept of state apart from the records that are stored within the stream. It also allows you to write loops to output more than one event per input event, and conditionals that allow you to decide whether to output an event at all.

The syntax for a FlexStream is as follows:

```
<FlexStream ... istream="name of input streams separated by spaces" >
{InputWindow | OutConnection}*
(Column)+
[Local]
[Method]+
[Timer]
</FlexStream>
```

Attributes:

istream	This attribute specifies the name of the input streams.

The Method elements specify how to compute events from other input events. There must be one Method for each input stream. Method elements have the syntax

```
<Method
[name="optional text"]
stream="name of input stream">
...
</Method>
```

Attributes of the Method elements:

name	This attribute specifies the name of the Method. It's meant purely for documentation purposes.
stream	This attribute specifies the name of the input stream. When an event arrives on that particular input stream, the block of SPLASH code within the Method is executed.

FlexStreams may also have a Timer element. This is a block of SPLASH code that executes periodically. Timer elements have the syntax

```
<Timer
interval="number of seconds">
...
</Timer>
```

Attributes of the Timer element:

interval	This attribute specifies the number of seconds between executions of the block. The default is 1 (second).

Example:

```
<FlexStream id="SomeCurrencies" store="store" istream="Currencies">
<Method name="inputMethod" stream="input">
if (getOpcode(Currencies) = update and
     Currencies.Currency = 'EUR')
   output Currencies;
</Method>
</FlexStream>
```

This is a FlexStream that forwards only those events whose `Currency` field is 'EUR' and is an update.

A FlexStream can be assigned to a Stateless Store. In this case, if a Method or Timer element attempts to output an event that is not an insert, the event will be rejected.

3.9.12. Pattern Stream

A Pattern Stream is a stream whose input comes from one or more streams, and whose output is generated by pattern-matching rules. The rules are written in a special pattern language that extends the SPLASH language. Patterns can, for instance, check whether events occur or do not occur in some time interval, and then send new events to downstream streams.

The syntax for a Pattern Stream is as follows:

```
<PatternStream ...
istream="name of input streams separated by spaces" >
{InputWindow | OutConnection}*
(Column)+
[Local]
[Pattern]*
</PatternStream>
```

Each Pattern element has the form

```
<Pattern [name="optional text"]>
...
</Pattern>
```

The text between `<Pattern>` and `</Pattern>` is a pattern written in the language described in Appendix C, *Pattern Matching Language* below. A Pattern Stream can have as many patterns as you want.

Attributes of the Pattern elements:

name	This attribute specifies a name for the pattern. It is used for documentation only.

Example:

```
<PatternStream id="PairTrades" store="store" istream="Trades">
<Column key="true" name="id" datatype="int32"/>
<Column key="false" name="Symbol1" datatype="string"/>
<Column key="false" name="Symbol2" datatype="string"/>
<Local>
int32 idloc := 0;
</Local>
<Pattern>
within 1 seconds
from Trades[Symbol='IBM'; Price=p] as trade1,
Trades[Symbol='MSFT'; Price=q] as trade2
on trade1 fby trade2
{
```

```
idloc := idloc + 1;
output [id = idloc; | Symbol1='IBM'; Symbol2='MSFT'];
}
</Pattern>
<Pattern>
within 5 seconds
from Trades[Symbol='CSCO'; Price=p] as trade1,
Trades[Symbol='LU'; Price=q] as trade2
on trade1 and trade2
{
idloc := idloc + 1;
output [id = idloc; | Symbol1='CSCO'; Symbol2='LU'];
}
</Pattern>
</PatternStream>
```

This Pattern Stream watches for possible pairs trading. It watches for two possible pairs trades, IBM® and Microsoft®, and Cisco and Lucent, in slightly different ways. The first detects whether an IBM trade is followed by a Microsoft trade within a one second interval. The second detects whether a Cisco and a Lucent trade happen in the same five second interval, in either order.

3.10. Best Practices When Writing an AleriML Data Model

Some suggestions:

- Use consistent and descriptive names when naming objects within the data model file.
- Coding the data model in incremental steps makes it easier to debug.
- Although Columns and ColumnExpressions can appear in any order, it improves readability to list the key fields first in alphabetical order followed by non-key fields in alphabetical order.
- Use the `valueInserted` or `any` function instead of the `max` or `min` function in aggregations. It's more efficient.
- Self-joins can be done by creating a Copy Stream and then joining it to the original.

Chapter 4. SPLASH Programming Language

This chapter describes the Streaming Platform LAnguage SHell (SPLASH) programming language used within FlexStreams and Pattern Streams.

The syntax of SPLASH is a combination of the expression language, including all of the functions described in Appendix B, *Data Types, Operators and Functions*, and a C-like syntax for blocks of statements. Just as in C, there are variable declarations within blocks, and statements for making assignments to variables, conditionals and looping. Other data types, beyond the scalar types described in Appendix B, *Data Types, Operators and Functions*, are also available within SPLASH, including types for records, collections of records and iterators over those records.

4.1. Preliminaries

Names and constants in SPLASH follow the conventions described in Chapter 1, *Authoring Preliminaries*. Comments can appear as blocks of text inside `/*-*/` pairs, or as line comments with `//`.

4.2. Variable and Type Declarations

SPLASH variable declarations resemble those in C: the type precedes the variable name(s), and the declaration ends in a semicolon. The variable can be assigned an initial value as well. Here are some examples of SPLASH declarations:

```
int32 a, r;
double b := 9.9;
string c, d := 'dd';
[ int32 key1; string key2; | string data; ] record;
```

The first three declarations are for scalar variables of types int32, double, and string. The first has two variables. In the second, the variable “b” is initialized to 9.9. In the third, the variable “c” is not initialized but “d” is. The fourth declaration is for a record with three columns. The key columns “key1” and “key2” are listed first before the `|` character; the remaining column “data” is a non-key column. The syntax for constructing new records is parallel to this syntax type.

The `typeof` operator provides a convenient way to declare variables. For instance, if `rec1` is an expression with type `[ int32 key1; string key2; | string data; ]` then the declaration

```
typeof(rec1) rec2;
```

is the same as the declaration

```
[ int32 key1; string key2; | string data; ] rec2;
```

SPLASH type declarations also resemble those in C. The `typedef` operator provides a way to define a synonym for a type expression. For instance, the declarations

```
typedef double newDoubleType;
typedef [ int32 key1; string key2; | string dataField; ] rec_t;
```

create new synonyms `newDoubleType` and `rec_t` for the double type and the given record type, respectively. Those names can then be used in subsequent variable declarations like

```
newDoubleType var1;
rec_t var2;
```

which improves the readability and the size of the declarations.

4.3. Data Structures

SPLASH has a rich set of data structures. This section describes those data structures.

4.3.1. Record Events

Record events — records with an associated operation like “insert” — can be created directly in SPLASH. We will use the word “record” interchangeably for “record event.”

The syntax of record types specifies the names and types of fields, and the key structure. For instance, the type

```
[ int32 key1; string key2; | string dataField; ]
```

describes records with key fields key1 and key2, of types int32 and string respectively, and the non-key field dataField of type string. The key fields appear before the “|” symbol.

The syntax of record values mirrors that of record types. Here, for example, is a record with the previous type:

```
[ key1 = 9; key2 = 'USD'; | string data = 'US Currency'; ]
```

The syntax of record values is fairly flexible. You can write the same record as

```
[ key1 = 9; key2 = 'USD' | string data = 'US Currency' ]
[ key1 = 9; key2 = 'USD' | string data = 'US Currency'; ]
```

eliding any semi-colon but those between `field = value`.

The operation of a new record value is insert. To change it, you use the `setOpcode` function, as in

```
setOpcode([ key1 = 9 | string data = 'US Currency' ], update)
```

Records with more fields can be used in a context expecting fewer fields; the extra fields get coerced away. Conversely, records with fewer fields can be used in a context expecting more fields; the missing fields are assumed to be null. For instance, if `var` is a variable of type

```
[ int32 key1; | string dataField; double otherData]
```

you can set

```
var := [key1 = 1; dataField = 'newdata'];
```

The record value will be implicitly cast to the right type, making key1 the key field and setting the other-Data field to null.

Operations on records:

Get a field	Get a field Syntax: `record.field` Type: The value returned has the type of the field. Example: `rec.data1`
Assign a field	Assign a field in a record. Syntax: `record.field := value` Type: The value must be a value matching the type of the field of the record. The expression returns a record. Example: `rec.data1 := 10`
`copyRecord`	Copies a record (deprecated; this function does nothing). Syntax: `copyRecord(`*`record`*`)` Type: The argument must be a record. The function returns a record. Example: `copyRecord(input)`
`getOpcode`	Gets the operation associated with a record. The operations are of type int32, and have the following meaning: • 1 means “insert” • 3 means “update” • 5 means “delete” • 7 means “upsert”(insert if not present, update otherwise) • 13 means “safe delete”(delete if present, ignore otherwise) Syntax: `getOpcode(`*`record`*`)` Type: The argument must be an event. The function returns an int32. Example: `getOpcode(input)`
`setOpcode`	Sets the operation associated with a record; the legal operations are listed above. Syntax: `setOpcode(`*`record, number`*`)` Type: The first argument must be a record, and the second an int32. The function returns the modified record. Example: `setOpcode(input,insert)`

4.3.2. XML Values

An XML value is a value composed of XML elements and attributes, where elements can contain other XML elements or text. XML values can be created directly or built by parsing string values. XML values cannot be stored in records, but can be converted to string representation and stored in that form.

You can also declare a variable of `xml` type, for example,

```
xml xmlVar;
```

and assign it to XML values.

Operations on XML values:

`xmlagg`	Aggregate a number of XML values into a single value. This can be used only in Aggregate Streams or with event caches (see below). Syntax: `xmlagg(xml value)` Type: The argument must be an XML value. The function returns an XML value. Example: `xmlagg(xmlparse(stringCol))`
`xmlconcat`	Concatenate a number of XML values into a single value. Syntax: `xmlconcat(xml value, ..., xml value)` Type: The arguments must be XML values. The function returns an XML value. Example: `xmlconcat(xmlparse(stringCol), xmlparse('<t/>'))`
`xmlelement`	Create a new XML data element, with attributes and XML expressions within it. Syntax: `xmlelement(name, [xmlattributes(string AS name ..., string AS name) ,] [xml value,...,xml value])` Type: The names must adhere to the conventions in Section 1.2, “Names”. The function returns an XML value. Example: `xmlelement(top, xmlattributes('data' as attr1), xmlparse('<t/>'))`
`xmlparse`	Convert a string to an XML value. Syntax: `xmlparse(string value)` Type: The argument must be a string value. The function returns an XML value. Example: `xmlparse('<tag/>')`
`xmlserialize`	Convert an XML value to a string. Syntax: `xmlserialize(xml value)` Type: The argument must be an XML value. The function returns a string Example: `xmlserialize(xmlparse('<t/>'))`

4.3.3. Vectors

A vector is a sequence of values, all of which must have the same type, with an ability to access elements of the sequence by an integer index. A vector has a size, from a minimum of 0 to a maximum of 2 billion entries. Vectors use semantics inherited from C: when accessing elements by index, index 0 is the first position in the vector, index 1 is the second, and so forth.

You can declare vectors in Global or Local blocks via the syntax

```
vector(valueType) variable;
```

For instance, you can declare a vector holding 32-bit integers like

```
vector(int32) pos;
```

Operations on vectors:

Create	Create a new empty vector. Syntax: `new vector(type)` Type: A vector of the declared type is returned. Example: `pos := new vector(int32);`
Get value by index	Get a value from the vector. If the index is less than 0 or greater than or equal to the size of the vector, return null. Syntax: `vector[index]` Type: The index must have type int32. The value returned has the type of the values held in the vector. Example: `pos[10]`
Assign a value	Assign a cell in the vector. Syntax: `vector[index] := value` Type: The index must have type int32, and the value must match the value type of the vector. The value returned is the updated vector. Example: `pos[5] := 3`
`size`	Returns the number of elements in the vector. Syntax: `size(`*`vector`*`)` Type: The argument must be a vector. The value returned has type int32. Example: `size(pos)`
`push_back`	Inserts an element at the end of the vector and returns the modified vector. Syntax: `push_back(`*`vector`*`, `*`value`*`)` Type: The second argument must be a value with the value type of the vector. The return value has the type of the vector. Example: `push_back(pos, 3)`
`resize`	Resize a vector, either removing elements if the vector shrinks, or adding null elements if the vector expands. Syntax: `resize(`*`vector`*`, `*`newsize`*`)`

Type: The second argument must have type int32. The return value has the type of the vector.

Example: `resize(vec1, 2)`

You can also iterate through all the elements in the vector (up to the first null element) using a “for” loop. See Section 4.4.6, “For Loops” for more information.

4.3.4. Dictionaries

A dictionary is a data structure that associates keys to values. They are called maps in C++ and Java, arrays in AWK, and association lists in LISP, so it's an old and very familiar data structure.

You can declare a dictionary in a Global or Local block via the syntax

```
dictionary(keyType, valueType) variable;
```

For instance, if you have an input stream called "input", you could store an int32 for distinct records as

```
dictionary(typeof(input), int32) counter;
```

Only one value is stored per key. That means that it's important to understand what equality on keys means. For the simple data types, equality means the usual equality, for example, equality on int32 or on string values. For record types, equality means that the keys match (the data fields and operation are ignored).

Operations on dictionaries:

Create	Create a new empty dictionary. Syntax: `new dictionary(type, type)` Type: A vector of the declared type is returned. Example: `d := new dictionary(int32, string);`
Get value by key	Get a value from the dictionary by key. If there is no such key in the dictionary, return null. Syntax: `dictionary[key]` Type: The key must have the type of the keys of the dictionary. The function returns a value of the type of the values held in the dictionary. Example: `counter[input]`
Assign a value by key	Associate a value to a key in the dictionary. Syntax: `dictionary[key] := value` Type: The key and value must match the key type and value type of the dictionary. The function returns the updated dictionary. Example: `counter[input] := 3`

Remove a key/value pair	Remove a key, and its associated value, from the dictionary. Syntax: `remove(dictionary, key)` Type: The key must match the key type of the dictionary. The function returns an int32: 0 if the key was not present, and 1 otherwise. Example: `remove(counter, input)`
Clear a dictionary	Remove all key/value pairs from the dictionary. Syntax: `clear(dictionary)` The function returns the cleared dictionary. Example: `clear(counter)`
Test for emptiness	Test a dictionary for emptiness. Syntax: `empty(dictionary)` The function returns an int32: 1 if the dictionary is empty, 0 if not empty. Example: `empty(counter)`

You can also iterate through all the key/value pairs in the dictionary using a “for” loop. See Section 4.4.6, “For Loops” for more information.

4.3.5. Streams

There are ways to access the records in input streams, using means similar to dictionaries, although one cannot change the records in an input stream.

Operations on streams:

Get value by key	Get a value from the stream by key. If there is no such key in the stream, return null. Syntax: `streamValue[ recordValue ]` Type: The key must have the record type of the stream. The operation returns a value of the record type of the stream. Example: `input_stream[ [k = 3; | ] ]` Note that the non-key fields of the argument do not matter; the operation will return a record with the current values of the non-key fields, if a record with the key fields exist. If a key field is missing from the argument, or the key field is null, then this operation will always return null. It doesn't make sense to compare key fields in the stream to null, since null is never equivalent to any value (including null).
Get value by match	Get a record from the stream that matches the given record. Unlike getting a value by key, there might be more than one matching record. If there is more than one matching record, one of the matching records is returned. If

there is no such match in the stream, null is returned.

Syntax: `streamValue{ recordValue }`

Type: The record must be consistent with the record type of the stream. The operation returns a value of the record type of the stream.

Example: `input_stream{ [ | d = 5 ] }`

You can use key and non-key fields in the record.

You can also iterate through all the records in a stream using a “for” loop. See Section 4.4.6, “For Loops” for more information.

4.3.6. Stream Iterators

Stream iterators are a means of explicitly iterating over all of the records stored in either one of the input streams, or in the stream itself. It's usually more convenient, and safer, to use the `for` loop mechanism described in Section 4.4.6, “For Loops”, but sometimes the extra flexibility of stream iterators is needed.

In FlexStreams, each block of code has implicit variables for streams and stream iterators. If an input stream, or the FlexStream itself, is named `Stream1`, there are variables `Stream1_stream` and `Stream1_iterator`.

Those variables can be used in conjunction with the following functions.

`deleteIterator` Releases the resources associated with an iterator.

Syntax: `deleteIterator(`*`iterator`*`)`

Type: The argument must be an iterator expression. The function returns a null value.

Example: `deleteIterator(input_iterator)`

Note:

Stream iterators are not implicitly deleted. If you don't delete them explicitly, all further updates to the stream may be blocked.

`getIterator` Get an iterator for a stream.

Syntax: `getIterator(`*`stream`*`)`

Type: The argument must be a stream expression. The function returns an iterator.

Example: `getIterator(input_stream)`

`getNext` Returns the next record in the iterator, or null if there are no more records.

Syntax: `getNext(`*`iterator`*`)`

Type: The first argument must be an iterator expression. The function returns a record, or “null” if there is no more data in the iterator.

Example: `getNext(input_iterator)`

`resetIterator`

Resets the iterator to the beginning.

Syntax: resetIterator(*iterator*)

Type: The argument must be an iterator expression. The function returns an iterator.

Example: `resetIterator(input_iterator)`

`setRange`

Sets a range of columns to search for. Subsequent getNext calls will return only those records whose columns match the given values.

Syntax: setRange(*iterator*, *fieldName*, ... *expr*...)

Type: The first argument must be an iterator expression; the next arguments must be the names of fields within the record; the final arguments must be expressions. The function returns an iterator.

Example:
`setRange(input_iterator,Currency,Rate,'EUR',9.888)`

`setSearch`

Sets values of columns to search for. Subsequent getNext calls will return only those records whose columns match the given values.

Syntax: setSearch(*iterator*, *number*, ... *expr*...)

Type: The first argument must be an iterator expression; the next arguments must be column numbers (starting from 0) in the record; the final arguments must be expressions. The function returns an iterator.

Example: `setSearch(input_iterator,0,2,'EUR',9.888)`

Note:

The `setSearch` function has been deprecated because it requires a specific layout of fields. It has been retained for backwards compatibility with existing models. When developing new models, use the `setRange` function instead.

4.3.7. Event Caches

An event cache holds a number of previous events for the input stream or streams to a derived stream. It is organized into buckets, based on values of the fields in the records. It's often used when vectors or dictionaries are not quite the right data structure.

You can define an event cache in a Local block. A simple event cache declaration is

```
eventCache(instream) e0;
```

This event cache holds all the events for an input stream “instream”. The default key structure of the input stream defines the bucket policy. That is, the buckets in this stream correspond to the keys of the input stream.

Suppose the input stream in this case has two fields, a key field k and a data field d. Suppose the events have been

```

<instream ALERI_OPS="i" k="1" d="10"/>
<instream ALERI_OPS="u" k="1" d="11"/>
```

```
<instream ALERI_OPS="i" k="2" d="21"/>
```

After these events have flowed in, there will be two buckets. The first bucket will contain the first two events, because these have the same key; the second bucket will contain the last event.

Event caches allow for aggregation over events. That is, the ordinary aggregation operations that can be used in Aggregate Streams can be used in the same way over event caches. The “group” that is selected for aggregation is the one associated with the current event. For instance, if a new event

```
<instream ALERI_OPS="u" k="1" d="12"/>
```

appears in this stream, then the expression `sum(e0.d)` will return 10+11+12=33. You can use any of the aggregation functions in Section B.9, “Aggregation Functions”, including `avg`, `count`, `max`, and `min`.

The next subsections describe the following options to the event cache type and the operations on event caches.

- `manual` and `auto` insertion
- Different keys to determine the buckets
- Size of buckets (by maximum time, number of events, or both) and policy (`jump` or `nojump`)
- Records instead of events (`coalesce`)
- Ordering the events in the buckets

4.3.7.1. Manual insertion

By default, every event that comes into a stream with an event cache gets put into the event cache. You can explicitly indicate this default behavior with the `auto` option, for example,

```
eventCache(instream, auto) e0;
```

You can also put events into an event cache if they are marked `manual`, for example,

```
eventCache(instream, manual) e0;
```

Use the function `insertCache` described below to do this.

4.3.7.2. Changing buckets

An event cache organizes events into buckets. By default, the buckets are determined from the keys of the input stream. You can change that default behavior to alternative keys, specifying other fields in square brackets after the name of the stream.

For example, specifying

```
eventCache(instream[d0,d1]) e0;
```

keeps buckets organized by distinct values of the d0 and d1 fields. To keep one large bucket of all events, write the following:

```
eventCache(instream[]) e0;
```

4.3.7.3. Managing the size of buckets

You can also manage the size of buckets in an event cache. That can often be important in controlling the use of memory.

You can limit the size of a bucket to the most recent events, by number of seconds, or by time:

```
eventCache(instream, 3 events) e0;
eventCache(instream, 3 seconds) e1;
```

You can also specify whether to completely clear the bucket when the size or time expires by specifying the `jump` option:

```
eventCache(instream, 3 seconds, jump);
```

The default is `nojump`.

All of these options can be used together. For example, this example clears out a bucket when it reaches 10 events (when the 11th event comes in) or when 3 seconds elapse.

```
eventCache(instream, 10 events, 3 seconds, jump);
```

4.3.7.4. Keeping records instead of events

You can keep records in an event cache, instead of distinct events for insert, update, and delete, by specifying the `coalesce` option:

```
eventCache(instream, coalesce) e0;
```

This option is most often used in conjunction with the next option, ordering.

4.3.7.5. Ordering

Normally, the events in a bucket are kept by order of arrival. You can specify a different ordering by the fields of the events. For instance, to keep the events in the bucket ordered by field d in descending order, write

```
eventCache(instream, d desc) e0;
```

You can order by more than one field. The following example orders the buckets by field d0 in descending order, then by field d1 in ascending order in case the d0 fields are equal.

```
eventCache(instream, d0 desc, d1 asc) e0;
```

4.3.7.6. Operations on Event Caches

`expireCache` Remove events from the current bucket that are older than a certain number of seconds.

Syntax: `expireCache(`*`events`*`, `*`seconds`*`)`

Type: The first argument must name an event cache variable. The second argument must be an int32. The function returns the event cache.

Example: `expireCache(events, 50)`

`insertCache` Insert a record value into an event cache.

Syntax: `insertCache(`*`events`*`, `*`record`*`)`

Type: The first argument must name an event cache variable. The argument must be a record type. The function returns the record inserted.

Example: `insertCache(events, inputStream)`

`keyCache` Select the current bucket in an event cache. Normally, the current input record selects the active bucket. You might want to change the current active bucket in some cases. For example, during the evaluation of the debugging expressions, there is no current input record and thus no bucket is set by default. The only way to set the bucket then is to do it manually using this function.

Syntax: `keyCache(`*`events`*`, `*`event`*`)`

Type: The first argument must name an event cache variable. The second argument must be a record type. The function returns the same record.

Example: `keyCache(ec1, rec)`

You can see more examples on the **sp_cli** man page in the *Guide to Programming Interfaces*.

4.4. Statements

SPLASH has statement forms for expressions, blocks, conditionals, output, “break” and “continue”, “while” and “for” loops, as well as blocks of statements.

4.4.1. Expression Statements

For instance, any expression can be turned into a statement by terminating the expression with a semicolon, as in

```
setOpcode(input, 3);
```

Since assignments are expressions, assignments can be turned into statements in the same way. For instance, the following statement assigns a string to a variable “address”:

```
address := '550 Broad Street';
```

4.4.2. Block Statements

Statements can be a sequence of statements, wrapped in braces, with optional variable declarations, as in

```
{
  double d := 9.99;
  record.b := d;
}
```

Variable declarations can be interspersed with statements, as in

```
{
  double pi := 3.14;
  print (string(pi));
  double e := 2.71;
  print (string(e));
}
```

4.4.3. Conditional Statements

Another form of statement is the conditional, which has the same syntax as C. For instance, you can write

```
if (record.a = 9)
  record.b := 9.99;
```

Conditionals may have optional “else” statements, as in

```
if (record.a = 9)
  record.b := 9.99;
else {
  double d := 10.9;
  record.b := d;
}
```

4.4.4. Output Statements

The `output` statement schedules an event to be sent to downstream streams, and also to be entered into the store of the stream:

```
output [k = 10; | d = 20;];
```

It is valid only in Flex and Pattern Streams.

Note:

Multiple output's can be done in processing an event; the outputs are collected as a transaction block. Similarly, if a Flex or Pattern Stream receives a transaction block, the entire transaction block is processed and all output is collected into another transaction block. This means that downstream streams, and the record data stored within the stream, are not changed until the entire event (single event or transaction block) is processed.

If a FlexStream is assigned to a Stateless Store, all attempts to `output` a non-insert are rejected.

4.4.5. While Statements

Another form of statement is “while” loops. Again, this has the same syntax as C. For instance, you can write

```
while (not(isnull(record))) {
  record.b := record.a + record.b;
  record := getNext(record_iterator);
}
```

4.4.6. For Loops

Loops are more often coded with “for” loops, which provide a convenient means of looping over some or all of the records in an input stream, or all of the data in a vector or dictionary.

To loop over every record in an input stream called “input”, write

```
for (record in input_stream) {
  ...
}
```

The variable `record` is a new variable; you can use any name here. The scope is the statement or block of statements in the loop; it has no meaning outside the loop. You can also set equality criteria in searching for records with certain values of fields, for example,

```
for (record in input_stream where c=10, d=11) {
  ...
}
```

which has the same looping behavior, except limited to the records whose c field is 10 and d field is 11. If you search on the key fields, the loop will run at most one time, but it will run extremely fast because it will use the underlying index of the stream.

To loop over the values in a vector “vec1”, write

```
for (val in vec1) {
 ...
}
```

where again, `val` is any new variable. The loop stops when the end of the vector is reached, or the value of the vector is null.

To loop over the values in a dictionary “dict1”, write

```
for (key in dict1) {
 ...
}
```

where again, `key` is any new variable. It's common, inside the loop, to use the expression `dict1[key]` to get the value held in the dictionary for that particular key.

4.4.7. Control Statements

Both `while` loops and `for` loops can be restarted or terminated, as in C. A `break` statement terminates the innermost loop; a `continue` statement starts the innermost loop over.

The `return` statement stops the processing and returns a value. This is most useful in SPLASH functions, described in the next section.

The `exit` statement stops the processing. This is most useful in processing an event in a FlexStream method, described in Section 4.6, "Using SPLASH within FlexStreams".

4.4.8. Switch Statements

The `switch` statement is a specialized form of conditional. For instance, you can write

```
switch(intvar*2) {
  case 0: print('case0'); break;
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

This statement prints "case0" if the value of `intvar*2` is 0, "case2" if the value of `intvar*2` is 2, and "default" otherwise. The `default` is optional. The expression inside the parentheses `switch(...)` must be of base type, and the expressions following the `case` keyword must have the same base type.

As in C and Java, the `break` is needed to skip to the end. For instance, if you leave out the `break` after the first `case`,

```
switch(intvar*2) {
  case 0: print('case0');
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

then the statement will print both "case0" and "case2" when `intvar*2` is 0.

4.5. Functions

You can write your own functions in SPLASH. They can be declared in Global blocks, for use by any stream, or Local blocks. A function can internally call other functions, or call themselves recursively.

The syntax of SPLASH functions resembles C. In general, a function looks like

```
type functionName(type1 arg1, ..., typen argn) { ... }
```

where each "type" is a SPLASH type, and each `arg` is the name of an argument. Within the `{...}` can appear any SPLASH statements. The value returned by the function is the value returned by the `return` statement within.

Here are some examples:

```
int32 factorial(int32 x) {
   if (x <= 0) {
     return 1;
   } else {
     return factorial(x-1) * x;
   }
 }
string odd(int32 x) {
   if (x = 1) {
      return 'odd';
   } else {
      return even(x-1);
   }
 }

string even(int32 x) {
   if (x = 0) {
     return 'even';
   } else {
    return odd(x-1);
   }
}
int32 sum(int32 x, int32 y) { return x+y; }
string getField([ int32 k; | string data;] rec) { return rec.data;}
```

The first function is recursive. The second and third are mutually recursive; unlike C, you do not need a prototype of the “even” function in order to declare the “odd” function. The last two functions illustrate multiple arguments and record input.

The real use of SPLASH functions is to define, and debug, a computation once. Suppose, for instance, you have a way to compute the value of a bond based on its current price, its days to maturity, and forward projections of inflation. You might write a function

```
double bondValue(double currentPrice,
                 int32 daysToMature,
                 double inflation)
{
  ...
}
```

and use it in many places within the data model.

4.6. Using SPLASH within FlexStreams

The following examples assume that there is a Source Stream with the declaration

```
<SourceStream id="inputStream" store="store">
  <Column key="true" name="a" datatype="int32" />
  <Column key="true" name="b" datatype="string" />
  <Column name="floatData" datatype="double" />
  <Column name="dateData"  datatype="date" />
</SourceStream>
```

FlexStreams use SPLASH declarations in the variables attribute, and SPLASH statements in Method elements. There are certain variables that are predefined in every Method. For each input stream, and the FlexStream itself, there are distinguished variables for records of that stream, the entire collection of records in that stream, and iterators over the stream. Suppose you have an input stream named “Currency”. There are implicit variables

`Currency`	for records in that stream
`Currency_old`	for previous records in that stream (more on this in a moment)
`Currency_stream`	for the collection of records
`Currency_iterator`	for iterators over that stream

For instance, you could write

```
Currency_iterator := getIterator(Currency_stream);
Currency := getNext(Currency_iterator);
```

in a FlexStream.

You can also access the data in the FlexStream itself. If the FlexStream is named “FS”, the variables are `FS` for records in that stream, `FS_stream` for the collection of records, and `FS_iterator` for iterators over that stream.

The first example illustrates the “output” statement and record expressions.

```
<FlexStream id="compute" store="store" istream="inputStream">
  <Column name="a" datatype="int32" key="true">
  <Column name="b" datatype="string">
  <Column name="floatData" datatype="double">
  <Column name="dateData" datatype="date">
  <Method name="inputMethod" stream="inputStream">
    output [a = inputStream.a + 1; |
            b = concat('hh', inputStream.b);
            floatData = 1.777;
            dateData = inputStream.dateData;];
  </Method>
</FlexStream>
```

This Method constructs a new record, adding 1 to the “a” column of the input record, and so on.

Notice the use of the variable `inputStream` within the expression. When an event arrives from one of the input streams, the Method whose “stream” attribute matches the input stream is called. The record is bound to the variable with the name of the input stream. In this example, when a record arrives on the stream called “inputStream”, the Method (the only one for this FlexStream) gets called with the record bound to the variable `inputStream`.

If the event is an update, the variable `inputStream_old` gets bound to the old record. That can be useful in tracking changes.

The next example illustrates the usage of a loop.

```
<FlexStream id="compute" store="store" istream="inputStream">
  <Column name="a" datatype="int32" key="true">
  <Column name="b" datatype="string">
  <Column name="floatData" datatype="double">
  <Column name="dateData" datatype="date">
  <Method name="inputMethod" stream="input">
    {
      int32 var;
      var := 0;
      while (var < 10) {
        output [a = var + 10 * input.a; |
                b = input.b;
                floatData = input.floatData;
                dateData = input.dateData;];
        var := var + 1;
      }
    }
  </Method>
</FlexStream>
```

For each input record, we output 10 records, each with a different value of the key column `a`.

The next example illustrates two features: the usage of more than one Method for more than one input stream and the usage of the “variables” attribute.

```
<FlexStream id="compute" store="store"
    istream="inputStream1 inputStream2">
  <Column name="a" datatype="int32" key="true">
  <Column name="b" datatype="string">
  <Column name="floatData" datatype="double">
  <Column name="dateData" datatype="date">
  <Local>
    int32 num1;
    int32 num2;
  </Local>
  <Method name="inputMethod1" stream="inputStream1">
    {
      if (isnull(num1)) num1 := 0;
      num1 := num1 + 1;
      output [a = num2; |
              b = inputStream1.b;
              floatData = inputStream1.floatData;
              dateData = inputStream1.dateData;];
    }
  </Method>
  <Method name="inputMethod2" stream="inputStream2">
    {
      if (isnull(num2)) num2 := 0;
      num2 := num2 + 1;
      output [a = num1; |
              b = inputStream2.b;
              floatData = inputStream2.floatData;
              dateData = inputStream2.dateData;];
    }
  </Method>
</FlexStream>
```

The variables declared in the variables attribute can be used in any Method. Here, they are used to communicate between the two Methods. The variables are initially set to `null`, which explains the need for the first “if” statements within each Method.

The last example shows that the FlexStream code can output a record with a completely different structure than the input stream.

```
<FlexStream id="compute" store="store" istream="inputStream">
  <Column name="a" datatype="int32" key="true">
  <Column name="b" datatype="string">
  <Column name="c" datatype="double">
  <Method name="inputMethod" stream="inputStream">
    {
      [ int32 a; | string b; double c;] rec;
      if (input.a <= 3) {
        rec := [a = input.a; | b = input.b;];
        rec.c := 8.88;
        setOpcode(rec,getOpcode(input));
        output rec;
      }
    }
  </Method>
</FlexStream>
```

Appendix A. Reserved Words

The following table provides a list of Reserved Words. Reserved Words are case insensitive.

AGGREGATE	AND	AS
ASC	AUTO	BEGIN
BREAK	BY	CASE
CAST	CLUSTER	CONNECTION
CONTINUE	COUNT	CREATE
DATA	DAY	DAYS
DECLARE	DEFAULT	DELETE
DESC	DISTINCT	DYNAMIC
ELSE	END	EVENTCACHE
EXIT	EXPIRES	EXTERNAL
FALSE	FBY	FIRST
FOR	FOREIGN	FOREIGNJAVA
FROM	FULL	FUNCTION
GENERATE	GRANT	GROUP
HASH	HAVING	HOUR
HOURS	HRS	IF
IN	INDEX	INNER
INSERT	INTERMEDIATE	IS
JOIN	KEY	LANGUAGE
LAST	LEFT	LIBRARY
LIKE	LOCAL	LOCATION
LOGSTORE	MATERIALIZED	MAX
MAXSIZE	MEMSTORE	MIN
MINUTE	MINUTES	MODULE
NAME	NEW	NOT
NTH	NULL	ON
OR	ORDER	OUTPUT
PATTERN	PRIMARY	PROGRAM
RANK	RECORDS	RETAIN
RETURN	RIGHT	SAFEDELETE
SEC	SECOND	SECONDARY
SECONDS	SELECT	SET
SETRANGE	STATELESSSTORE	STATIC
STORAGE	STORE	SUM
SWITCH	SYNC	TABLE
THEN	TIMES	TO
TOP	TREE	TRUE
TYPE	TYPEDEF	TYPEOF
UNION	UPDATE	UPSERT
VIEW	WHEN	WHERE
WHILE	WITHIN	XMLATTRIBUTES
XMLELEMENT		

Appendix B. Data Types, Operators and Functions

This section lists all the supported data types, operators, and functions. These apply regardless of the authoring environment. Any differences between the authoring environments are noted.

B.1. Data Types

The following table lists the data types supported by the Sybase Aleri Streaming Platform.

Data Type	Description
int32	32-bit integer
int64	64-bit integer
money	Fixed-point number. With the default money precision of 4 decimal digits, the range of values is -922,337,203,685,477.5808 through +922,337,203,685,477.5807.
double	Floating-point number (IEEE double precision)
date	Date/time (64-bit date and time field for 64-bit machines, and 32-bit field for 32-bit machines), represented as the number of seconds since the epoch (1970-01-01 00:00:00+00, or 1 January 1970 at midnight UTC)
timestamp	Date/time in milliseconds (64-bits), represented as the number of milliseconds since the epoch (1970-01-01 00:00:00.000+00, or 1 January 1970 at midnight UTC)
string	Variable length character string

Note:

The number of decimal digits in the “money” datatype can be changed with the moneyPrecision attribute in the Platform object (see Section 3.2, “Platform”). For instance, if moneyPrecision is set to 7, the range of values is -922,337,203,685.4775808 through +922,337,203,685.4775807.

All types other than string are considered numeric types. Arithmetic can be done on any numeric type. Rules, called casting in many programing languages, are also applied for changing one numeric type to another. These rules are as follows:

- Int32 values can be promoted to int64, money, date, timestamp or double values.
- int64 values can be promoted to date, timestamp or double values.
- date values can be promoted to timestamp or double values. In the case of promotion to timestamp values, the result is scaled. For example, the number of seconds is multiplied by 1000 to get the number of milliseconds.
- timestamp values can be promoted to double values.
- money values can be promoted to double values.

For example, data of type int32 will automatically be promoted to a double if it is added to a double. The additions to the usual rules for type promotion also apply to money and date values.

B.2. Opcodes/Constants

The following names, which are Opcodes and Constants, have default values for use in expressions.

`delete`	Name for the delete operation. Equivalent to the int32 value 5. Type: int32
`insert`	Name for the insert operation. Equivalent to the int32 value 1. Type: int32
`null`	Name for the null value. Type: any type
`safedelete`	Name for the safe delete operation (delete if present, ignore otherwise). Equivalent to the int32 value 13. Type: int32
`update`	Name for the update operation. Equivalent to the int32 value 3. Type: int32
`upsert`	Name for the upsert operation (insert if not present, update if present). Equivalent to the int32 value 7. Type: int32

B.3. Special Columns

All streams have two special columns. The first special column, rowid of type `int64`, holds an integer that is a unique value for every row of the stream. The second special column, rowtime of type date, stores the time (in seconds) at which the row was last modified (or created). Expressions may refer to these columns in the same way they do to any other column.

B.4. Nulls and Error Handling

The operators and functions below, unless otherwise noted, return null if any of their arguments are null.

Some operators and functions can cause run-time errors (for example, divide-by-zero). In these cases, since all operators and functions are involved in the computation of an event, the server stops processing the event and logs an error.

B.5. Arithmetic Operators

Arithmetic operators can be applied to all numeric types. Arithmetic operators can be used with mixed numeric types, with implicitly promoted values. For example, if a date value is divided by an int32, the result gets promoted to the larger of the two types.

The following table describes the supported arithmetic operators.

Operator	**Meaning**	**Example Usage**
+	Addition	3+4
-	Subtraction	7-3
*	Multiplication	3*4
/	Division	8/2
%	Modulus (remainder)	8%3
^	Exponent	4^3

Operator	Meaning	Example Usage
-	Change signs	-3

Division and modulus can cause run-time errors if the divisor is 0. If the argument is a double, the server logs a “Floating-point exception” error. If the arguments are not doubles, the server logs a “Divide-by-zero” error.

B.6. Comparison Operators

The following table describes the supported comparison operators.

Standard Operator	Meaning	Example Usage
=	Equal to	a0 = a1
!=	Not equal to	a0 != a1
>	Greater than	a0 > a1
>=	Greater than or equal to	a0 >= a1
<	Less than	a0 < a1
<=	Less than or equal to	a0 <= a1
in	Member of a list of values	a0 in (a1, a2, a3)

B.7. Boolean Operators

The following table describes the supported Boolean operators. These operators expect to operate on values of type int32 and return values of type int32. These functions have the same names regardless of the authoring environment.

Operator	Meaning	Example Usage
and	Return 1 if all values are not equal to 0, and 0 otherwise.	(a < 10) and (b > 12)
not	Return 1 if all values are equal to 0, and 0 otherwise.	not (a = 5)
or	Return 1 if any of the values are not equal to 0, and 0 otherwise.	(b = 8) or (b = 6)

B.8. Arithmetic Functions

`abs` Returns the absolute value of a number.

Syntax: `abs(`*`number`*`)`

Type: The argument must be a numeric type and the function returns a numeric value.

Example: `abs(-88.76)` returns 88.76.

`acosine` Returns the inverse cosine of a number between -1 and 1. If the argument is invalid (outside the range -1 and 1), the server logs a “Floating-point exception” error.

Syntax: `acosine(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `acosine(0.0)` returns 1.570796.

`asine` Returns the inverse sine of a number between -1 and 1. If the argument is invalid (outside the range -1 and 1), the server logs a “Floating-point exception” error.

Syntax: `asine(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `asine(1.0)` returns 1.570796.

`atangent` Returns the inverse tangent of a number between -1 and 1. If the argument is invalid (outside the range -1 and 1), the server logs a “Floating-point exception” error.

Syntax: `atangent(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `atangent(1.0)` returns 0.785398.

`cbrt` Returns the cube root of a number. If the argument is invalid, the server logs a “Floating-point exception” error.

Syntax: `cbrt(`*`number`*`)`

Type: The argument must be a numeric type and the function returns a double.

Example: `cbrt(1000.00)` returns 10.0.

`ceil` Rounds a number up.

Syntax: `ceil(`*`number`*`)`

Type: The argument must be of type double and the function returns a double.

Example: `ceil(100.20)` returns 101.0.

`cosine` Returns the cosine of a number expressed in radians. If the argument is invalid, the server logs a “Floating-point exception” error.

Syntax: `cosine(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `cosine(0.5)` returns 0.87758.

`exp` Returns the value of e (the base of natural logarithm, 2.78128) raised to the power of a number. If the argument is invalid, the server logs a “Floating-point exception” error.

Syntax: `exp(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `exp(2.0)` returns 7.3890.

`floor` Rounds a number down.

Syntax: `floor(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `floor(100.20)` returns 100.0.

`ln`
: Returns the natural logarithm of a number. If the argument is invalid (for example, less than 0), the server logs a “Floating-point exception” error.

Syntax: `ln(`*`number`*`)`

Type: The argument must be of type double and the function returns a double.

Example: `ln(2.718281828...)` returns 1.0.

`log`
: Returns the base 10 logarithm of a number. If the argument is invalid (for example, less than 0), the server logs a “Floating-point exception” error.

Syntax: `log(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `log(100)` returns 2.0.

`round`
: Round the first value to the number of digits specified by the second value.

Syntax: `round(`*`number, digits`*`)`

Type: Both arguments must be of type double and the function also returns a double.

Example: `round(66.778, 1)` returns 66.8.

`sine`
: Returns the sine of a number expressed in radians. If the argument is invalid, the server logs a “Floating-point exception” error.

Syntax: `sine(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `sine(0.5)` returns 0.4794255.

`sqrt`
: Returns the square root of a number. If the argument is invalid (for example, less than 0), the server logs a “Floating-point exception” error.

Syntax: `sqrt(`*`number`*`)`

Type: The argument must be of type double or money; the function returns a double.

Example: `sqrt(100.0)` returns 10.0.

`tangent`
: Returns the tangent of a number expressed in radians. If the argument is invalid, the server logs a “Floating-point exception” error.

Syntax: `tangent(`*`number`*`)`

Type: The argument must be of type double and the function also returns a double.

Example: `tangent(0.5)` returns 0.546302.

B.9. Aggregation Functions

These functions operate on multiple records to calculate one value from a group of values. They can be used in three places: inside Aggregate Streams, within SQL queries with a “group by” clause, and in conjunction with the event cache type (see Section 4.3.7, “Event Caches” below).

`any`	Returns some value in the group of values; the choice of value is dependent on the implementation.
	Syntax: `any(`*`expr`*`)`
	Type: The argument can be of any type and the function returns the same data type as the argument.
	Example: `any(v.Currency)`
`avg`	Computes the average of the non-null values in the group. The expression can be of any numeric type except "date" or "timestamp".
	Syntax: `avg(`*`number`*`)`
	Type: The argument can be of any numeric type except "date" or "timestamp" and the function returns the same data type as the argument.
	Example: `avg(v.Shares)`
`count`	Counts the number of non-null values in the group.
	Syntax: `count(`*`expr`*`)`
	Type: The argument can be of any type but the function always returns an int32.
	Example: `count(v.Price)`
`count_distinct`	Counts the number of distinct non-null values in the group.
	Syntax: `count_distinct(`*`expr`*`)`
	Type: The argument can be of any type but the function always returns an int32.
	Example: `count_distinct(v.Price)`
`count(*)`	Counts the number of records in the group.
	Syntax: `count(*)`
	Type: The function always returns an int64.
	Example: `count(*)`
`first`	Returns the first value from the group of values.
	Syntax: `first(`*`expr`*`)`
	Type: The argument can be of any type and the function returns the same data type as the argument.
	Example: `first(v.Price)`
`last`	Returns the last value from the group of values.

	Syntax: `last(`*`expr`*`)`
	Type: The argument can be of any type and the function returns the same data type as the argument.
	Example: `last(v.Price)`
`lwm_avg`	Returns the linearly weighted moving average for the group of values.
	Syntax: `lwm_avg(`*`number`*`)`
	Type: The argument can be of any numeric type and the function returns the same data type as the argument.
	Example: `lwm_avg(v.Price)`
`max`	Returns the maximum value from the group of values.
	Syntax: `max(`*`expr`*`)`
	Type: The argument can be of any type and the function returns the same data type as the argument.
	Example: `max(v.Price)`
`min`	Returns the minimum value from the group of values.
	Syntax: `min(`*`expr`*` )`
	Type: The argument can be of any type and the function returns the same data type as the argument.
	Example: `min(v.Price)`
`nth`	Returns the nth value from the group of values. The first argument determines the value to be returned (0 for the newest).
	Syntax: `nth(`*`number, expr`*`)`
	Type: The first argument must be an int32 but the second argument can be of any type. The function returns the same data type as the argument.
	Example: `nth(4,v.Price)`
`recent`	Returns the most recent non-null value in the group of values.
	Syntax: `recent(`*`expr`*`)`
	Type: The argument can be of any type and the function returns the same data type as the argument.
	Example: `recent(v.Price)`
`stddev_pop`	Returns the population standard deviation for the group of values.
	Syntax: `stddev_pop(`*`number`*`)`
	Type: The argument can be of any numeric type and the function returns a double.

Example: `stddev_pop(v.Price)`

`stddev_samp` Returns the sample standard deviation for the group of values.

Syntax: `stddev_samp(`*`number`*`)`

Type: The argument can be of any numeric type except date and timestamp, and the function returns a double.

Example: `stddev_samp(v.Price)`

`sum` Computes the sum of the non-null values in the group.

Syntax: `sum(`*`number`*`)`

Type: The argument can be of any numeric type except date and timestamp, and the function returns the same data type as the argument.

Example: `sum(v.Shares)`

`valueInserted` Returns a value from the group based on the last row inserted into that group.

Syntax: `valueInserted(`*`expr`*`)`

Type: The argument can be of any type and the function returns the same data type as the argument.

Example: `valueInserted(v.Shares)`

B.10. String Functions

These functions operate on one or more string arguments.

`concat` Concatenates the given string arguments into a single string and returns that value.

Syntax: `concat(`*`string1`*`, ... `*`stringn`*`)`

Type: The function takes one or more string arguments and returns a string. Literal text must be enclosed in single quotation marks.

Example: `concat('MSFT', '_NYSE')` returns 'MSFT_NYSE'.

`length` Returns the length of a string passed.

Syntax: `length(`*`string`*`)`

Type: The argument must be a string and the function returns an int32

Example: `length('abc')` returns 3.

`like` Determines whether a string matches a pattern string. Returns 1 if the string matches the pattern, and 0 otherwise. The pattern can contain wildcards: '_' matches a single arbitrary character; '%' matches 0 or more arbitrary characters.

Syntax: `like(`*`string`*`, `*`pattern`*`)`.

In SQL, the infix notation can also be used: sourceString like patternString.

Type: Both arguments must be strings and the function returns an int32.

Example: `like('MSFT', 'M%T')` returns 1.

`lower` Returns a new string where all the characters of the input string are in lower case.

Syntax: `lower(`*`string`*`)`

Type: The argument must be a string and the function also returns a string

Example: `lower('This Is A Test')` returns 'this is a test'.

`ltrim` Trims spaces from the left of a string.

Syntax: `ltrim(`*`string`*`)`

Type: The argument must be a string and the function also returns a string

Example: `ltrim(' sourcestring')` returns 'sourcestring'.

`patindex` Determines the position of the nth occurrence of a pattern within a source string. The pattern can contain wildcards: "_" matches a single arbitrary character; "%" matches 0 or more arbitrary characters. If fewer than n instances of the pattern are found in the string, the function returns -1.

Syntax: `patindex(`*`string`*`, `*`pattern`*`, `*`number`*`)`

Type: The first two arguments must be strings and the third must be an int32. The function returns an int32.

Example: `patindex('longlonglongstring', 'long', 2)` returns 4 (the first position in the string is 0). And `patindex('longstring', 'long', 2)` returns -1.

`replace` In the first string argument, replace all occurrences of the second string argument with the third string.

Syntax: `replace(`*`target`*`, `*`substring`*`, `*`repstring`*`)`

Type: The function takes three string arguments and returns a string.

Example: `replace('NewAmsterdam', 'New', 'Old')` returns 'OldAmsterdam'.

`right` Returns the rightmost characters of a string.

Syntax: `right(`*`string`*`, `*`number`*`)`

Type: The first argument must be a string and the second must be an int32. The function returns a string.

Example: `right('sourcestring', 6)` returns 'string'.

`rtrim` Trims spaces from the right of a string.

Syntax: `rtrim(`*`source`*`)`

Type: The argument must be a string; the function returns a string.

Example: `rtrim('sourcestring ')` returns 'sourcestring'.

`substr`
: Computes a substring from a string, given a start position and the number of characters to be taken. The starting position is 0.

  Syntax: `substr(`*`string`*`, `*`position`*`, `*`number`*`)`

  Type: The first argument must be a string, the second and third arguments must be int32. The function returns a string.

  Example: `substr('thissubstring', 4, 3)` returns 'sub'.

`upper`
: Returns a new string where all the characters of the input string are in upper case.

  Syntax: `upper(`*`string`*`)`

  Type: The argument must be a string and the function also returns a string

  Example: `upper('This Is A Test')` returns 'THIS IS A TEST'.

B.11. Date and Time Functions

These functions operate on one or more date arguments.

`date`
: Converts a date value to an int32 with digits "yyyymmdd".

  Syntax: `date(`*`dateVal`*`)`

  Type: The argument must be a date but the function returns an int32.

  Example: `date(v.TradeTime)`

`datename`
: Converts a date value to a string of the form "yyyy-mm-dd".

  Syntax: `datename(`*`dateVal`*`)`

  Type: The argument must be a date but the function returns a string.

  Example: `datename(v.TradeTime)`

`datepart`
: Returns an int32 representing a portion of the date:

  Syntax: `datepart(`*`portion`*`, `*`dateVal`*`)`

  where portion can be one of the following strings:

  - The year, if the string is `yy` or `yyyy`.
  - The month, if the string is `mm` or `m`.
  - The day of the year, if the string is `dy` or `y`.
  - The day of the month, if the string is `dd` or `d`.
  - The day of the week, if the string is `dw`.
  - The hour, if the string is `hh`.

- The minute, if the string is `mi` or `n`.
- The second, if the string is `ss` or `s`.

Type: The first argument must be a string and the second be a date. The function returns an int32.

Example: `datepart('ss', v.TradeTime)` returns the seconds portion of the date value.

`sysdate`

Returns the current system date as a date value.

Syntax: `sysdate()`

Type: The function has no arguments and returns a date.

Example: `sysdate()`

`systimestamp`

Returns the current system date as a timestamp value.

Syntax: `systimestamp()`

Type: The function has no arguments and returns a date.

Example: `systimestamp()`

`totimezone`

Converts a date from the time zone specified in the second argument (a string) to the corresponding time in the time zone specified in the third argument. Time zone values are taken from the industry-standard TZ database. See Appendix G, *List of Time Zones* for a complete list of legal time zones.

Syntax: `totimezone(`*`date`*`, `*`fromTimeZone`*`, `*`toTimeZone`*`)`

Type: The first argument must be a date; the second and third must be strings. The function returns a date.

Example: `totimezone(v.TradeTime, 'GMT', 'EDT')` converts the time from Greenwich Mean Time to Eastern Daylight Time.

`trunc`

Truncates the time portion of a date to 00:00:00 and returns the new date value.

Syntax: `trunc(`*`date`*`)`

Type: The argument must be a date and the function returns a date.

Example: `trunc(v.TradeTime)`

B.12. Calendar Functions

Calendar functions supply the name of a calendar file as their first argument. A calendar file is a text file in the format:

```
weekendStart <integer>
weekendEnd <integer>
holiday yyyy-mm-dd
holiday yyyy-mm-dd
...
```

In the `weekendStart` and `weekendEnd` lines, the integer represents the day of the week: with Monday=0, Tuesday=1, ..., Saturday=5, and Sunday=6. The file can have as many "holiday" lines as needed. Lines beginning with "#" are ignored.

The following is an example of a legal calendar file:

```
# Sybase calendar data for US 1983
weekendStart 5
weekendEnd 6
holiday 1983-02-21
holiday 1983-04-01
holiday 1983-05-30
holiday 1983-07-04
holiday 1983-09-05
holiday 1983-11-24
holiday 1983-12-26
```

Calendar files are loaded and cached on demand by the Sybase Aleri Streaming Platform. If changes occur in any of the calendar files, a command must be sent to the Sybase Aleri Streaming Platform to refresh the cached calendar data. See **sp_cli** for a description of the **refresh_calendars** command.

The calendar functions are:

`business` Determines the next business day from a date value and an offset. The offset can be any negative or positive integer. Negative integers return previous business days. The Sybase Aleri Streaming Platform will return null if the offset is 0 (it shouldn't be 0) and will log a message.

Syntax: `business(`*`calendarFile`*`, `*`dateVal`*`, `*`offset`*`)`

Type: The first argument must be a string, the second argument a date, and the third an int32. The function returns a date.

Example: `business('/aleri/cals/us.cal',v.TradeTime, 1)`

`businessDay` Determines if a date value falls on a business day (a day that is neither a weekend nor a holiday). It returns 1 if true, and 0 otherwise.

Syntax: `businessDay(`*`calendarFile`*`, `*`dateVal`*`)`

Type: The first argument must be a string, and the second argument a date. The function returns an int32.

Example: `businessDay('/aleri/cals/us.cal',v.TradeTime)`

`weekendDay` Determines if a date value falls on a weekend.

Syntax: `weekendDay(`*`calendarFile`*`, `*`dateVal`*`)`

Type: The first argument must be a string, and the second argument a date. The function returns an int32.

Example: `weekendDay('/aleri/cals/us.cal',v.TradeTime)`

B.13. Type Conversion Functions

These functions are used to convert data from one data type to another. All of these functions except the “cast” function operate on a single argument.

`cast` Converts a value of one numeric type to other numeric type.

Syntax: `cast(`*`type, number`*`)`

Type: The type must be one of the following values: int32, int64, double, money, date, or timestamp

The expression must be a type that can be cast to the specified type. It is legal to cast expressions of any type except a string type. Casting from larger types to smaller ones may cause overflow. Casting from decimal types (like double or money) to non-decimal types (like int32) truncates the decimal portion.

Example: `cast(timestamp, v.TradeTime)`

`dateInt` Converts a date value to an int32 that represents the number of seconds since 1970-01-01 00:00:00 UTC (the Epoch).

Syntax: `dateInt(`*`dateVal`*`)`

Type: The argument must be a date; the function returns an int32.

Example: `dateInt(v.TradeTime)`

`int32` Converts a string to an int32.

Syntax: `int32(`*`string`*`)`

Type: The argument must be a string but the function returns an int32.

Example: `int32('9988')` returns 9988.

`intDate` Converts an int32 representing the number of seconds since 1970-01-01 00:00:00 UTC (the Epoch) to a date.

Syntax: `intDate(`*`number`*`)`

Type: The argument must be an int32 and the function returns a date.

Example: `intDate(1)` returns a date value, for which ISO string representation is 1970-01-01 00:00:01.

`real` Converts a string to a double.

Syntax: `real(`*`string`*`)`

Type: The argument must be a string and the function returns a double. The string must not contain commas.

Example: `real('77.8866')` returns 77.8866.

`string` Converts a value of any type to an equivalent string representation.

Syntax: `string(`*`value`*`)`

Type: The argument can have any type but the function returns a string.

Example: `string(4512)` returns '4512'.

`undate` Convert a string in the ISO date of the form "yyyy-mm-dd hh:MM:ss" into a date value.

Syntax: `undate(`*`string`*`)`

Type: The argument must be a string; the function returns a date.

Example: `undate('2003-06-14 13:15:00')`

B.14. Null Handling and Rank Functions

`firstnonnull` Returns the first non-null value from a list of arguments. If all values are null, it returns null.

Syntax: `firstnonnull(`*`expr1, ... exprn`*`)`

Type: All the arguments must have the same type; the function returns a value of that type.

Example: `firstnonnull(v.Price, 0.0)` returns `v.Price` if the value is non-null and 0.0 otherwise.

`ifnull` Returns the first non-null value from a list of arguments. If all values are null, it returns null. (This function behaves exactly like `firstnonnull`.)

Syntax: `ifnull(`*`expr1, ... exprn`*`)`

Type: All the arguments must have the same type; the function returns a value of that type.

Example: `ifnull(v.Price, 0.0)` returns `v.Price` if the value is non-null and 0.0 otherwise.

`isnull` Returns 1 if the argument is null, and 0 otherwise.

Syntax: `isnull(`*`expr`*`)`

Type: The argument can be of any type and the function returns an int32.

Example: `isnull('hello')` returns 0.

`rank` Returns the position of the row in the current group, starting from position 0. This function is useful in GroupFilter (Group Having clause in SQL Authoring) expressions only.

Syntax: `rank()`

Type: This function has no arguments. It returns an int32.

Example: `rank() > 3` returns 1 for the first three rows in a group and 0 for all other rows.

B.15. User-Defined Functions

Functions, written in a language like C/C++ or Java, can be called from the Sybase Aleri Streaming Platform. Refer to Appendix D, *User-Defined Functions* below for more information.

`foreign` Calls a function from a shared library.

Syntax: `foreign(`*`sharedLibrary`*`, `*`function`*`, `*`type`*`, `*`expr`*`, ... `*`expr`*`)`

Type: The first two arguments must be identifiers (wrapped in double quotes if they contain special characters like periods); the third argument must be the name of a type; the rest must be arguments to the function. The function returns a value of the return type specified.

Example: `foreign(&quot;distance.so&quot;, distance, double, u.a, u.b)`

`foreignJava` Calls a static Java function.

Syntax: `foreignJava(`*`className`*`, `*`function`*`, `*`type`*`, `*`expr`*`, ... `*`expr`*`)`

Type: The first two arguments must be identifiers (wrapped in double quotes if they contain special characters like periods); the third argument must a string specifying the type of the function; the rest must be arguments to the function. The function returns a value of the return type specified.

Example: `foreignJava(Funs, distance, '(DD)D', u.a, u.b)`

B.16. Print

This function prints values.

`print` Print strings on the standard output; this is especially useful in debugging.

Syntax: `print(string1, ..., stringn)`

Type: The arguments must be strings; the function returns null.

Example: `print('here: ', string(i))`.

B.17. Assignment

Variables can be assigned using “:=” (note the difference in syntax from Java and C/C++'s operator "="). The type and value of the expression is the type and value of the expression assigned.

For instance,

```
var1 := v.Price
```

returns the value of `v.Price` after setting the value of the variable `var1`.

You can also assign directly to columns if the record is bound to a variable (see Section 4.3.1, “Record Events” for more information). For instance, you can write

```
record.address := '550 Broad Street';
```

to assign the column “address” to the string. Note that this will change the record in place, so that the old value of the column will not be available.

You can also use “++” and “--” as in C, with a variable. The expression `++v` increments the value of `v`, and returns the new value (pre-increment). The expression `v++` increments the value of `v`, and returns the old value (post-increment). Similarly, `--v` is pre-decrement, and `v--` is post-decrement.

B.18. Sequencing

Expressions can be combined with semicolons, wrapped in parentheses, to be evaluated in order. The type and value of the expression is the type and value of the last expression.

For instance,

```
(var1 := v.Price; 0.0)
```

returns 0.0 after setting the value of the variable `var1`.

B.19. Conditional Expressions

Conditional expressions begin with the keyword `case`:

```
case
when expression then expression
...
else expression
end
```

For example,

```
case
when (v.Price < 100.0) then 1.0
when (v.Price > 200.0) then 2.0
else v.Price
end
```

returns 1.0 when the price is less than 100.0, 2.0 when the price is greater than 200.0, and the price otherwise.

The types of the conditional expressions must be int32 and the types of the branches must match.

B.20. External Data Functions

`getData` Get records from an external database via an SQL statement. The records are stored in a vector; see Section 4.3.3, “Vectors” below for a description of vectors.

Syntax: `getData(`*`vector`*`, `*`data location`*`, `*`SQL`*`, `*`expr1`*`, ... `*`exprn`*`)`

Type: The first argument must be a vector of records, the second a string representing the DataLocation, the third an SQL string, and the last arguments strings. The last arguments are put into holes, marked by a “?” character, in the SQL statement. The function returns the vector. See Section 3.7, “DataLocation” for a description of DataLocations.

Example: `getData(v,'ora','select ID,? from Data1','A')` gets records from a table “Data1” in a DataLocation named “ora”, and puts records with the two selected fields “ID” and “A” into the vector “v”.

B.21. Unique Value Functions

`uniqueId` Generate a new int64 value. This value starts at 0, and is different for every call (even from different streams).

Syntax: `uniqueId()`

Type: The function returns an int64.

Example: `uniqueId()`

Appendix C. Pattern Matching Language

Patterns are the building blocks of Pattern Streams. They have the following syntax:

```
within ...
from ...
on ...
(computational clause)
```

The following sections describe each of the clauses.

C.1. Within clause

The text following `within` must be a specification of the interval in which the events occur. It can be specified in seconds, minutes, or hours. For example, you can write

```
within 5 seconds ...
```

You can also use `minutes` or `hours` instead of `seconds` (with the obvious meaning) for convenience.

C.2. From clause

The `from` clause specifies the content of events to be matched. Each event is separated from the next by a comma, and each event has the form

```
streamName[fieldName={name|literal}; ...;
           fieldName={name|literal}] as eventName
```

The `streamName` component must be the name of one of the input streams of the Pattern Stream. The `eventName` gives a name for this particular event, for use in the `on` clause. Each `fieldName` should be the name of a column in that stream. You don't have to name all of the columns of the input stream within the `[...]`, only the ones you care about.

The power of patterns comes from the use of literals and names within the `[...]`. Specifying a literal means the field must have a particular value. Specifying a name allows you to use the value of the field in other patterns. For instance, if the from clause looks like

```
from Trades[Symbol='CSCO'; Price=p] as trade1,
     Trades[Symbol='LU'; Price=q] as trade2
```

it means that the Symbol field in the first event must be 'CSCO', and the Symbol field in the second event must be 'LU'. The variable `p` takes on the value of the Price field in the first event. Similarly, the variable `q` takes on the value of the Price field in the second event.

When the same name is used in different event patterns, the values must match. For instance, if the patterns are changed to

```
from Trades[Symbol='CSCO'; Price=p] as trade1,
     Trades[Symbol='LU'; Price=p] as trade2
```

the two events must have the same value in their Price field. If the names are not the same, the values may be the same or different. In other words, nothing is implied by different variable names.

There are some restrictions. You cannot use the same name in the same event, for example,

```
from Trades[Symbol='CSCO'; Price=p; LastPrice=p] as trade1
```

is not allowed. Also, there is no way to check for anything but equality between events. Other checks, like less-than or not-equal, can be encoded in the computational clause in SPLASH.

You can also check the event to see what kind of operation it is, by using the special field `ALERI_OPS`. You can check this only for equality to a literal or constant, and not assign it to a name. For example,

```
from Trades[ALERI_OPS=insert;Symbol='CSCO'; Price=p] as trade1
```

is legal, but

```
from Trades[ALERI_OPS=opc;Symbol='CSCO'; Price=p] as trade1
```

is not.

C.3. On clause

The `on` clause specifies the temporal relationships between events. You can use `and`, `or`, and `not` in the `on` clause, and the special operation `fby` (the “followed-by” operation), in combination with the event names from the `from` clause.

For instance, writing

```
event1 fby event2
```

means that the pattern match succeeds if event1 is followed by an event2 (that match the case in the `from` clause). Other non-matching events may happen in between.

The other boolean operations on events have the standard meaning. For instance, writing

```
event1 and event2
```

means that the pattern match succeeds if event1 and event2 happen (in either order).

C.4. Computational clause

The final clause is a computational clause. It must be a SPLASH statement or block, and should probably use the `output` statement form. See Chapter 4, *SPLASH Programming Language* for a description of the full language.

C.5. Examples

It's often easiest to see how to use a language with examples. Here are four examples of patterns in the pattern matching language.

The first example checks to see whether a broker sends a buy order on the same stock as one of his or her customers, then inserts a buy order for the customer, and then sells that stock. It creates a “buy ahead” event when those actions have occurred in that sequence.

```
within 5 minutes
from
  BuyStock[Symbol=sym; Shares=n1; Broker=b; Customer=c0] as Buy1,
  BuyStock[Symbol=sym; Shares=n2; Broker=b; Customer=c1] as Buy2,
  SellStock[Symbol=sym; Shares=n1; Broker=b; Customer=c0] as Sell
on Buy1 fby Buy2 fby Sell
{
  if ((b = c0) and (b != c1)) {
   output [Symbol=sym; Shares=n1; Broker=b];
  }
}
```

This example checks for three events, one following the other, using the `fby` relationship. Because the same variable `sym` is used in three patterns, the values in the three events must be the same. Different variables might have the same value, though (for example, `n1` and `n2`.) It outputs an event if the Broker and Customer from the Buy1 and Sell events are the same, and the Customer from the Buy2 event is different.

The next example shows Boolean operations on events. The rule describes a possible theft condition, when there has been a product reading on a shelf (possibly through RFID), followed by a non-occurrence of a checkout on that product, followed by a reading of the product at a scanner near the door.

```
within 12 hours
from
  ShelfReading[TagId=tag; ProductName=pname] as onShelf,
  CounterReading[TagId=tag] as checkout,
  ExitReading[TagId=tag; AreaId=area] as exit
on onShelf fby not(checkout) fby exit
output [TagId=t; ProductName=pname; AreaId=area];
```

The next example shows how to raise an alert if a user tries to log in to an account unsuccessfully three times.

```
within 5 minutes
from
  LoginAttempt[IpAddress=ip; Account=acct; Result=0] as login1,
  LoginAttempt[IpAddress=ip; Account=acct; Result=0] as login2,
  LoginAttempt[IpAddress=ip; Account=acct; Result=0] as login3,
  LoginAttempt[IpAddress=ip; Account=acct; Result=1] as login4
on (login1 fby login2 fby login3) and not(login4)
output [Account=acct];
```

People wishing to break into computer systems often scan a number of TCP/IP ports for an open one, and attempt to exploit vulnerabilities in the programs listening on those ports. Here’s a rule that checks whether a single IP address has attempted connections on three ports, and whether those have been followed by the use of the “sendmail” program.

```
within 30 minutes
from
  Connect[Source=ip; Port=22] as c1,
  Connect[Source=ip; Port=23] as c2,
  Connect[Source=ip; Port=25] as c3
  SendMail[Source=ip] as send
on (c1 and c2 and c3) fby send
output [Source=ip];
```

Appendix D. User-Defined Functions

Application developers can develop their own functions in C++, Java or off-the-shelf functions and connect them to the Sybase Aleri Streaming Platform via the User-Defined Function Interface. These external functions can be invoked within an expression in a Sybase data model.

The following section describes the process of developing and applying user-defined functions.

D.1. User-Defined Functions in C/C++

To create a usable user-defined functions in C/C++:

1. Write the function.
2. Build a shared library.
3. Write the call to the function within the model.

During the execution of the model, the Sybase Aleri Streaming Platform links to this library at run-time and calls the function as desired.

To make the process clear, this section develops a C++ function for computing Euclidean distance in three-dimensional space. The example comprises three simple steps:

- Write the function. Refer to Section D.1.1, "Write a User-Defined Function" for more information.
- Compile the function into a shared library. Refer to Section D.1.3, "Compile a User-Defined Function" for more information.
- Write the appropriate expression with the model to call the function. Refer to Section D.1.4, "Call a User-Defined Function" for more information.

D.1.1. Write a User-Defined Function

In C++, the function to compute distance is:

```
#include math.h
double distance(int numvals, double * vals)
{
  double sum = 0.0;
  for (int i=0; i<numvals; i++)
  sum += vals[i] * vals[i];
  return sqrt(sum);
}
```

To adapt this function for use in the Sybase Aleri Streaming Platform, the following requirements must be taken into consideration:

- All external functions must conform to the same interface. This means that the arguments passed to a user-defined function cannot be of type `double`, or any other specific C/C++ type. Instead, arguments are drawn from a structured type that includes all possible types understood by the Sybase Aleri Streaming Platform internally. This type, called "DataValue", is found in the `DataTypes.hpp` header file and currently has the following definition:

```
struct DataValue {
  union
  {
    int16_t int16v;
    int32_t int32v;
    int64_t int64v;
    money_t moneyv;
    double doublev;
    time_t datev;
    timestampval_t timestampv;
    const char *  stringv;
    void * objectv;
  } val;
  bool null;
};
```

Note:

The boolean flag is "null" within this structure. Functions must be aware of the fact that values may not be actual values but might be "null".

- The internal processing engine within the Sybase Aleri Streaming Platform is a bytecode stack machine that keeps the top of the stack in a special location. Therefore, a function must split the array of arguments into two:

  - A pointer to the top of the stack. In C, this is a value of type `(DataValue *)`.
  - An array of the rest of the arguments. In C, this is a value of type `(DataValue *)`.

  The Sybase Aleri Streaming Platform writes the return value of the function into the top of the stack.

- If the function allocates memory (by calling `malloc` or `calloc`), it must record the allocated memory so that it can be released later. A final argument to the function, a vector of `(void *)`, is used to record the allocated memory.

- Error codes: The function might need to return an error code. User-defined functions must return a value of type `int32_t`; usually this will be "NO_ERROR", which is a predefined value of 0.

  Thus, the interface to a user-defined function is:

```
int32_t ForeignFunction(int numargs,
        DataTypes::DataValue * top,
        DataTypes::DataValue * nextArgs,
        std::vector<void *> & arena)
```

The code for the distance function example is:

```
#include math.h
#include <vector>
#include “Row.hpp”
#include “DataTypes.hpp”
extern “C” int32_t distance(int numargs,
    DataTypes::DataValue * top,
    DataTypes::DataValue * nextArgs,
    std::vector<void *> & arena)
{
  double sum = 0.0;
  if (numargs <= 0) {
    top->null = false;
    top->val.double = 0.0;
    return 0;
  }
  if (top->null) return 0;
  double dist = top->val.doublev * top->val.doublev;
  for (int i=numargs-2; i>=0; i--) {
    if ((nextArgs + i)->null) {
      top->null = true;
      break;
    }
    dist +=(nextArgs + i)->val.doublev *(nextArgs + i)->val.doublev;
  }
  top->val.double = sqrt(dist);
  return 0;
}
```

The “extern” declaration is necessary within C++ to ensure that the function has the same name within the shared library and not the "mangled" C++ name.

D.1.2. A Second Example

This option pricing calculation example, based on the Binomial Model, illustrates how your code can access the arguments:

```
#include <math.h>
#include <float.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <vector>
#include "Row.hpp"
#include "DataTypes.hpp"
double opval_ambin(double S, double X,
                   double r, double sigma,
                   double hcost, double t,
                   int isPut, int steps)
/* Dictionary
   S = spot price; X = exercise price;
   r = interest rate; sigma = volatility;
   hcost = holding cost (risk-free rate for stocks)
   t = time to maturity;
   isPut = is the option a put? (0=call,
      1=put);
   steps = no steps in binomial tree
 */
{
  // code omitted
  ...
}
extern “C” int32_t coambin(int numargs,
    DataTypes::DataValue * top,
    DataTypes::DataValue * nextArgs,
    std::vector<void *> & arena)
{
  double S,X,r,sigma,hcost,t;
  int isPut,steps;

  // error and null checking
  if (numargs <= 7) {
    top->null = false;
    top->val.doublev = 0.0;
    return 0;
  }
  if (top->null) return 0;

  // get arguments
  S = (nextArgs+0)->val.doublev;
  X = (nextArgs+1)->val.doublev;
  r = (nextArgs+2)->val.doublev;
  sigma = (nextArgs+3)->val.doublev;
  hcost = (nextArgs+4)->val.doublev;
  t = (nextArgs+5)->val.doublev;
  isPut = (nextArgs+6)->val.int32v;
  steps = (top)->val.int32v;  //Last argument is the TOP of the stack

  // call the function
  top->val.doublev = opval_ambin(S,X,r,sigma,hcost,t,isPut,steps);
  return 0;
}
```

The first argument to the function is in `(nextArgs+0)`, the second is in `(nextArgs+1)`, and so forth. The top of the stack contains the last argument to the function, reflecting the order of evaluation of the arguments.

D.1.3. Compile a User-Defined Function

After writing the function, you must compile it into a shared library. Here is an example using the gcc compiler:

```
gcc -fPIC -shared -m64 -I.. -c -o distance.o distance.cpp
gcc -fPIC -shared -m64 distance.o -o distance.so
```

This creates a shared library named `distance.so`.

D.1.4. Call a User-Defined Function

After your code has been compiled, the function can be used in expressions with the `foreign` function:

```
foreign(&quot;distance.so&quot;, distance, double,u.a,u.b,u.c)
```

You may need to give a more complete path to the shared library.

Note:

When searching for shared libraries (.dll files), Windows® checks the path of the application (sp/sp-opt). If the .dll file is not found in that directory, other directories are searched, culminating in the directories specified in the `PATH` environment variable. More information may be found on the Microsoft web site.

D.2. User-Defined Functions in Java

To build user-defined functions in Java, you must

1. Write the function.
2. Compile the function (either into a `.class` file or a `.jar` file).
3. Write the call to the function within the model.

During the execution of the model, the Sybase Aleri Streaming Platform links to this library at run-time and calls the function as desired.

To make the process clear, this section develops some simple Java functions. The example comprises four steps:

1. Write the function. Refer to Section D.2.1, “Write User-Defined Functions in Java” for more information.
2. Compile the function. Refer to Section D.2.2, “Compile User-Defined Functions in Java” for more information.
3. Write the appropriate expression with the model to call the function. Refer to Section D.2.3, “Call User-Defined Functions in Java” for more information.
4. Start the Sybase Aleri Streaming Platform with the appropriate parameters. Refer to Section D.2.4, “Link User-Defined Functions in Java” for more information.

D.2.1. Write User-Defined Functions in Java

The Sybase Aleri Streaming Platform can call static Java functions defined in any class. Here, for instance, is a class "Functions" that defines a number of callable functions:

```
public class Functions {
  public static int intFun0() { return 172836; }
  public static int intFun1(int i,int j) { return i+j; }
  public static long longFun0() { return 967346; }
  public static long longFun1(long i,long j) { return i+j;}
  public static double doubleFun0() { return 10.7152; }
  public static double doubleFun1(double i, double j) { return i+j; }
  public static String stringFun0() { return "hij"; }
  public static String stringFun1(String i) { return i; }
}
```

The Sybase Aleri Streaming Platform supports arguments and return values of the Java types:

- int (32-bit integers)
- long (64-bit integers)
- double (double-precision floating point numbers)
- String (character strings)

D.2.2. Compile User-Defined Functions in Java

Next, the functions must be compiled into a Java class. For instance, you might use

```
javac -d /home/aleriusr/java/lib Functions.java
```

which compiles the Java code into a class file and writes that file in the directory `/home/aleriusr/java/lib`.

You can also create Java archives (`.jar`) files of classes, and refer to those in the classpath below as normal.

D.2.3. Call User-Defined Functions in Java

To use the Java function he function can be used in Sybase Aleri Streaming Platform expressions. For example, the expression

```
foreignJava(Functions,intFunction1,'(II)I',1,2)
```

calls the function `intFunction1` in the class `Functions` on two int32 arguments, and returns and int32.

The third argument to **foreignJava** is a string, a representation of the type of the function being called. You can obtain this string via the `javap` executable shipped with most versions of Java. For instance, if the class file is located in `/home/aleriusr/java/lib`, then

```
javap -s -classpath /home/aleriusr/java/lib Functions
```

produces

```
public class Functions extends java.lang.Object{
public Functions();
  Signature: ()V
public static int intFun0();
  Signature: ()I
public static int intFun1(int, int);
  Signature: (II)I
public static long longFun0();
  Signature: ()J
public static long longFun1(long, long);
  Signature: (JJ)J
public static double doubleFun0();
  Signature: ()D
public static double doubleFun1(double, double);
  Signature: (DD)D
public static java.lang.String stringFun0();
  Signature: ()Ljava/lang/String;
public static java.lang.String stringFun1(java.lang.String);
  Signature: (Ljava/lang/String;)Ljava/lang/String;
}
```

The lines beginning with “Signature” give the types to be used in the third argument. Here, “I” denotes a 32-bit integer, “J” a 64-bit integer, “D” a double, and “Ljava/lang/String;” a string.

Note:

Unlike C/C++ external functions, the interface for Java functions does not permit null values to be passed to the functions. You must handle null values explicitly, and not pass them to Java functions or unexpected results will occur.

D.2.4. Link User-Defined Functions in Java

The Sybase Aleri Streaming Platform has a default Java runtime environment built into it. All you need to do to “link” the Java code into your application is start the server with a special flag:

```
sp -j /home/aleriusr/java/lib ...
```

The `-j` option specifies the classpath for the Java virtual machine.

Note:

The Java runtime included with the Sybase Aleri Streaming Platform is Sun's Java 1.5. If your code needs a more recent version of Java (for example, Java 1.6), you can set the special environment variable `ALERI_SP_JAVA_HOME` to the location of the appropriate Java virtual machine shared library (usually `libjvm.so` on Linux® or Solaris® and `jvm.dll` on Windows). For instance,

```
export ALERI_SP_JAVA_HOME=/usr/bin/java/jre/lib/libjvm.so
```

sets the variable on a Linux or Solaris machine in the shell. You must then run the server in this environment.

Appendix E. Aleri Metadata Streams

Certain metadata streams are automatically created by the Sybase Aleri Streaming Platform. These streams hold information about the running data model. Metadata streams can be queried and subscribed to, but no stream in the data model may have a metadata stream as its input.

Metadata streams have the special reserved names. No other objects may use these reserved names. In general, all the names starting with `Aleri_` are reserved. Metadata streams also store their records in a special store called `AleriMetadataStore`. No other streams may use this store.

E.1. Aleri_Config

Aleri_Config contains the current AleriML configuration of the Sybase Aleri Streaming Platform. It has two string columns, key and value, and exactly one row (more rows may be added in the future). The key column contains 'XML' and the value column contains the text of the current AleriML configuration. This row gets updated if the Sybase Aleri Streaming Platform's configuration changes dynamically.

E.2. Aleri_Streams

Aleri_Streams contains information about all streams. It has the following columns:

Column	**Type**	**Description**
user_name	string	Currently hardcoded as `user`. In the future this will be the owner's username.
stream_name	string	Name of the stream described by this row.
handle	int64	The stream's handle (numeric id).

When a stream gets deleted by a Dynamic Modification, its row is removed. When a stream gets created by a Dynamic Modification, its row is added. When a stream gets dynamically modified in an incompatible way, its old row gets deleted and a new row (with new handle value) inserted. The compatible dynamic changes of the streams have no effects on their rows, except for renaming. The renaming still shows as a deletion and insertion but the stream handle stays the same.

Each insertion or deletion of a stream is sent as a separate transaction. When a stream is changed in an incompatible way, its old row is deleted and new one inserted. These are two separate transactions. For more information, see the *Administrator's Guide.*

E.3. Aleri_Tables

Aleri_Tables contains information about both source and derived streams. It has the following columns (which are taken from names in PostgreSQL):

Column	**Type**	**Description**
relname	string	Name of the stream described by this row.
username	string	Currently hardcoded as “user”, in the future will be the owner user's name.
relkind	string	Currently unused, empty.
remarks	string	The stream's handle (numeric id), formatted as a decimal number in the ASCII string.

When a stream gets deleted by a Dynamic Modification, its row is removed. When a stream gets created by a Dynamic Modification, its row is added. When a stream gets dynamically modified in an incompatible way, its old row gets deleted and the new row, with new handle value, inserted. The compatible dynamic changes of the streams have no effects on their rows, except for renaming. The renaming still shows as a deletion followed by an insertion, but the stream handle stays the same.

Each insertion or deletion of a stream is sent as a separate transaction. When a stream is changed in an incompatible way, the deletion of the old row and the insertion of the new version are separate transactions. For more information, see the *Administrator's Guide.*

E.4. Aleri_Columns

Aleri_Columns contains information about all columns of all streams. It has the following columns (which are taken from names in Postgres):

Column	Type	Description
usename	string	Currently hardcoded as "user". In future releases, the value will be the owner's username. (Note: the name of the column is "usename", not "username").
relname	string	Name of the stream that contains the column described by this row.
attname	string	Name of the column described by this row.
attypid	int32	The PostgreSQL value representing the type of this column. The possible values are: int32 = 23, int64 = 20, money = 701 (same as double), double = 701, date = 1114, timestamp = 1114 (same as date), string = 1043.
typname	string	Currently empty.
attnum	int32	Position of this column in the row definition, starting from zero.
attlen	int32	Currently unused, set to zero.
atttypmod	int32	Currently unused, set to zero.
attnotnull	string	Currently unused, set to zero.
relhasrules	string	Currently empty.
relkind	string	Currently empty.

When a stream gets deleted by a Dynamic Modification, the corresponding rows are removed. When a stream gets created by a Dynamic Modification, the corresponding rows are added. When a stream gets dynamically modified in an incompatible way, its old rows get deleted and the new rows for the new columns inserted. The compatible dynamic changes of the streams have no effects on their rows.

Each insertion or deletion of a stream is sent as a separate transaction. When a stream is changed in an incompatible way, its old records are deleted and new ones inserted, as two separate transactions. For more information see the *Administrator's Guide.*

E.5. Aleri_KeyColumns

Aleri_KeyColumns contains information about the key columns of all the streams. It has the following columns:

Column	Type	Description
table	string	Name of the stream owning the column described by this row.
field	string	Name of the column described by this row.
type	int32	The PostgreSQL value representing the type of this column. The possible values are: int32 = 23, int64 = 20, money = 701 (same as double), double = 701, date = 1114, timestamp = 1114 (same as

Column	Type	Description
		date), string = 1043.
type_name	string	Currently empty.
field_length	int32	Currently unused, set to 0.

When a stream gets deleted by a Dynamic Modification, the corresponding rows are removed. When a stream gets created by a Dynamic Modification, the corresponding rows are added. When a stream gets dynamically modified in an incompatible way, its old rows get deleted and the new rows (for the new columns) inserted. The compatible dynamic changes of the streams have no effects on their rows.

Each insertion or deletion of a stream is sent as a separate transaction. When a stream is changed in an incompatible way, the deletion of the old records and the insertion of the new one are two separate transactions. For more information, see the *Administrator's Guide*.

E.6. Aleri_Clients

Aleri_Clients contains information about all the currently active gateway client connections. It has the following columns:

Column	Type	Description
handle	int64	An unique integer id of the connection.
user_name	string	The username used for login on this connection. When a connection is first created, its username is NULL. After login, the row gets updated with the username used for logging in.
ip	string	The address of the client machine, as a string.
host	string	The symbolic host name of the client machine, if available. If the host name is not available, the value is the IP address.
port	int32	The TCP port number from which the connection originates.
login_time	timestamp	Time when the connection was accepted (not authenticated), in GMT.
conn_tag	string	The user-set symbolic connection tag name (see the option `-m` of **sp_subscribe** and **sp_upload**). If not set by the user, is NULL.

E.7. Aleri_Subscriptions

Aleri_Subscriptions contains the information about all the currently active subscriptions. It has the following columns:

Column	Type	Description
stream_handle	int64	The handle of the stream subscribed to (as in Aleri_Streams).
conn_handle	int64	The handle of the connection subscribed to the stream (as in Aleri_Clients).

This stream tracks the subscriptions and unsubscriptions done in every possible way. If a connection is dropped, it's considered unsubscribed from everything to which it was subscribed.

E.8. Aleri_Subscriptions_Ext

Aleri_Subscriptions_Ext contains the information about all the currently active subscriptions, in a denormalized but more convenient format. It has the following columns:

Column	Type	Description
stream_handle	int64	The handle of the stream subscribed to (as in Aleri_Streams).
conn_handle	int64	The handle of the connection subscribed to the stream (as in Aleri_Clients).
stream_name	string	Name of the stream. If a stream is dynamically renamed, this value will change.
stream_user	string	The username of the owner of the stream (as in Aleri_Streams).
subscriber_user	string	Login name of the user account that owns this subscription.
ip	string	Address of the client machine, as a string.
host	string	Symbolic host name of the client machine, if available. If the host-name is not available, the value is the IP address.
port	int32	The TCP port number from which the connection owning this sub-scription originates.
login_time	timestamp	Time when the connection owning this subscription was accepted, in GMT.

Note:

In some situations, this metadata stream may be slightly out of sync. For example, if a stream is deleted dynamically, all subscriptions to this stream are deleted too. You might see the row for the subscription updated, with null stream name and owner, before the row is eventually de-leted.

E.9. Aleri_Connectors

Aleri_Connectors contains information about all the InConnections and OutConnections defined in the Sybase Aleri Streaming Platform. The words “connector” and “connection” are often used interchange-ably as synonyms. The word "connection" may also be used to describe the client connections to the Sy-base Aleri Streaming Platform. gateway (through the pub/sub API). To reduce confusion, the word "con-nector" is used in this metadata stream's name. It has the following columns:

Column	Type	Description
name	string	An unique name of the connector, as defined in the model.
stream	string	Name of the stream on which this connector is defined.
type	string	Connector type, the same as the "type" attribute of this connector's DataLocation.
input	int32	"1" for InConnection, "0" for OutConnection.
ingroup	string	The StartUp group where this connector belongs.
state	string	The state of connector. One of: READY — Ready to be started. INITIAL — Performing start-up and initial loading. CONTINU-OUS — Continuously receiving real-time data. IDLE — Not currently receiving the data but attempting to re-connect to the data source or sink. DONE — No more input or output will follow, the connector

Column	Type	Description
		thread is about to exit. DEAD — The connector thread exited. The connector will stay in this state until explicitly requested to re-start.
total_rows	int64	Total number of data records recognized in the input data stream by an InConnection, or the number of data records received by an Out-Connection from the platform.
good_rows	int64	Number of data records successfully processed.
bad_rows	int64	Number of data records that experienced errors.

The fields total_rows, good_rows, and bad_rows are updated once in a few seconds, to reduce the over-head.

E.10. Aleri_RunUpdates

Aleri_RunUpdates delivers notifications of changes in the state of debugging. The notifications are sent only when the Sybase Aleri Streaming Platform is in trace mode. It is not a “real stream” in the sense that its store is always empty, and only updates are sent. It has the following columns:

Column	Type	Description
key	string	Type of the update
value	int32	Some integer value associated with the update
stream	string	If the update notifies of an event related to some individual stream, it contains the name of the stream. Otherwise NULL.
info	string	Some additional string information associated with the update. The format of this information depends on the type of the update.

The following types of updates are currently sent:

Key	Value	Stream	Description
`TRACE`	0 | 1	(none)	The trace mode has changed: enabled (1) or disabled (0).
`RUN`	0 | 1	(none)	The platform has paused (0) or continued running (1).
`STEP`	<count>	(none)	The Sybase Aleri Streaming Platform was single-stepped, manually or automatically. The value contains the number of the steps made. No specifics are provided about which streams were stepped.
`BREAK`	<bp-id>	<stream-name>	A breakpoint with ID <bp-id> has been triggered on the stream <stream-name>. These updates may come either before or after the corresponding update "RUN 0".
`NOBREAK`	<bp-id>	<stream-name>	A breakpoint with ID <bp-id> on the stream <stream-name> had its leftToTrigger count decreased, but it didn't trigger yet.

Key	Value	Stream	Description
EXCEP-TION	(none)	<stream-name>	An exception has happened on the stream <stream-name>. These updates may come either before or after the corresponding update "RUN 0".
REQUES-TEXIT	(none)	(none)	A request to shut down the Sybase Aleri Streaming Platform has been received.
EXIT	(none)	(none)	All the user streams have exited. The Sybase Aleri Streaming Platform is about to complete the shutdown.

E.11. Aleri_ClockUpdates

Aleri_ClockUpdates delivers notifications of changes in the logical clock of the Sybase Aleri Streaming Platform. It is not a “real stream” in the sense that its store is always empty, and only updates are sent. It has the following columns:

Column	Type	Description
key	string	Type of update, currently the only type is "CLOCK"
rate	double	Rate of the logical clock relative to the real time
time	double	The current time, in seconds since the UNIX® epoch
real	int32	Real time flag: 1 if the logical clock runs in real time (that is, matching the system time of the machine where the Sybase Aleri Streaming Platform runs), 0 if at varied rate or time.
stop_depth	int32	How many times the clock has been stopped recursively, or in other words how many times the clock resume would have to be called to actually resume the flow of time; when the clock is running, the stop depth is 0
max_sleep	int32	The period of time, in real milliseconds, that guarantees that all the sleepers discover the changes in the clock rate or time

E.12. Aleri_Streams_Monitor

Aleri_Streams_Monitor contains information about the performance of streams. Monitoring data is only available if the Sybase Aleri Streaming Platform was started with monitoring option **-t**. **Aleri_Clients_Monitor** contains basic information about the connected clients, but performance-related fields are only populated with the monitoring option.

This stream has the following columns:

Column	Type	Description
stream	string	Name of the stream.
cpu_pct	double	CPU usage in percent by this stream's thread in the time period since the last update.
trans_per_sec	double	The stream's performance in transactions per second, in the time period since the last update.
rows_per_sec	double	The stream's performance in rows per second, in the time period since the last update.
inc_trans	int64	Number of transactions processed since the last update.

Column	Type	Description
inc_rows	int64	Number of rows processed since the last update.
queue	int32	Current input queue size. A high size indicates that this stream can't process the records fast enough. This means that either this stream or one of its output streams is a bottleneck. The bottleneck can be located by looking for a stream with high input queue size but with all its output streams having low input queue sizes.
store_rows	int64	Current number of records in stream's store.
last_update	date	The time of the current update.
sequence	int64	The sequence number of the current update.
post-ing_to_client	int64	The numeric handle of the client connection, where this stream is trying to post the data at the moment. Most of the time it will contain -1, meaning “not trying to post right now”. This column may be useful for analyzing the situations when the Sybase Aleri Streaming Platform becomes unresponsive. If a client is not reading the data posted to it, its gateway queue will overflow, and any stream posting to it will become stuck. This effect will propagate throughout the Sybase Aleri Streaming Platform and will have all the streams involved show the low CPU usage and high queue size.

E.13. Aleri_Clients_Monitor

Aleri_Clients_Monitor contains information about the performance of all the currently active gateway client connections. For convenience, it is denormalized and contains a copy of data from **Aleri_Clients_Monitor**. Monitoring data is only available if the Sybase Aleri Streaming Platform was started with monitoring option **-t**. **Aleri_Clients_Monitor** contains basic information about the connected clients, but performance-related fields are only populated with the monitoring option. **Aleri_Clients_Monitor** has the following columns:

Column	Type	Description
handle	int64	An unique integer id of the connection.
user_name	string	The username used for login on this connection. When a connection is first created, its username is NULL. After login, the row gets updated with the username used for logging in.
ip	string	The address of the client machine, as a string.
host	string	The symbolic host name of the client machine, if available. If the host name is not available, the value is the IP address.
port	int32	The TCP port number from which the connection originates.
login_time	timestamp	Time when the connection was accepted (not authenticated), in GMT.
conn_tag	string	The user-set symbolic connection tag name (see the option `-m` of **sp_subscribe** and **sp_upload**). If not set by the user, is NULL.
cpu_pct	double	CPU usage in percent by this client's gateway thread in the time period since the last update.
last_update	date	The time of the current update.
subscribed	int32	Flag, showing whether this client is subscribed to any streams (1) or not (0).
sub_trans_per_sec	double	The client's performance in transactions per second received by the client, in the time period since the last update. For this purpose the envelopes and any service messages are also counted as transac-

Column	Type	Description
		tions.
sub_rows_per_sec	double	The client's performance in data rows per second received by the client, in the time period since the last update.
sub_inc_trans	int64	Number of transactions/envelopes/messages received by the client since the last update.
sub_inc_rows	int64	Number of data rows received by the client since the last update.
sub_total_trans	int64	Total number of transactions/envelopes/messages received by the client.
sub_total_rows	int64	Total number of data rows received by the client.
sub_dropped_rows	int64	Total number of data rows dropped in the gateway because the client did not read them fast enough (for lossy subscriptions).
sub_accum_size	int32	For the pulsed subscriptions, the current number of rows collected in the accumulator, to be sent in the next pulse.
sub_accum_ops	int32	Reserved for the future. The intent is to provide for the pulsed subscriptions the number of operations applied to the accumulator since the last pulse. It may differ from the accumulator size, if multiple operations become collapsed in the accumulator. Currently is a placeholder with the value of -1.
sub_queue	int32	Number of the rows queued for transmission to the client. This is just the “proper queue” part. The total number of records buffered consists of **sub_accum_size**, **sub_queue** and **sub_work_queue**.
sub_queue_fill_pct	double	Current **sub_queue** in percent relative to the queue size limit. If the queue size reaches this limit (100%), any future attempts to post data to this client will block, propagating the flow control back.
sub_work_queue	int32	Number of the rows for transmission to the client that are being transferred from the “proper queue” to the socket buffer. At this point the rows may get regrouped by envelopes.
pub_trans_per_sec	double	The client's performance in transactions per second sent by the client, in the time period since the last update. For this purpose the envelopes and any service messages are also counted as transactions.
pub_rows_per_sec	double	The client's performance in data rows per second sent by the client, in the time period since the last update.
pub_inc_trans	int64	Number of transactions/envelopes/messages sent by the client since the last update.
pub_inc_rows	int64	Number of data rows sent by the client since the last update.
pub_total_trans	int64	Total number of transactions/envelopes/messages sent by the client.
pub_total_rows	int64	Total number of data rows sent by the client.
pub_stream_id	int64	The numeric id of the stream to which the client is trying to publish the data at the moment. Most of the time it will contain -1, meaning “not trying to publish right now”. This column may be useful for analyzing situations when the Sybase Aleri Streaming Platform becomes unresponsive.

The data provided by this stream may be used to analyze performance issues with the clients.

High CPU usage means that the client has subscribed to a large amount of data, and its gateway connection may have become the bottleneck. If the client has subscribed to multiple streams, then on an SMP machine the bottleneck may be removed by splitting the subscriptions into two connections.

If the CPU usage is low but the subscription queue size is high then the client is not reading the data as fast as the Sybase Aleri Streaming Platform sends it. Such a client slows down the whole Sybase Aleri Streaming Platform. Perhaps the client has to be optimized. Another option would be to change its subscription mode to lossy or pulsed.

Appendix F. Data Location Descriptions, Parameters, Limits

The following list of connectors has parameter descriptions with basic, advanced and mandatory usage requirements, defaults and known limitations. For more information on the framework to add a connector not included in the set of built-in connectors that come with the Sybase Aleri Streaming Platform, see the *Guide to Programming Interfaces* .

You should note for connectors that support discovery, when a source stream is created via the connector discovery mechanism, the schema is defined, but key columns are not chosen.

F.1. ActivFinancial Inbound Plug-in

The ActivFinancial Inbound Plug-in connector works with the Activ Adapter which connects to the Activ Content Gateway to receive real-time Level 1 and/or Level 2 market data. This connector supports discovery. Activ connector can be configured on any source stream as an inbound data location.

This connector is listed as **activInPlugin** in the Aleri Studio's Data Location list. You must use this connector with Aleri Activ version 1.0.2 or later. The Adapter must be installed per Adapter Guide directions. Plug-in connectors are started on the same machine as the Sybase Aleri Streaming Platform which is controlled by the Remote Execution dialog.

See Aleri Activ documentation for more information on installing and configuring Adapters. Please contact a Sybase sales representative if you are interested in obtaining an Activ Adapter.

The "discover" gesture from the Aleri Studio activates a wrapper script which passes Aleri Studio-editable parameters. If the *`Discover Path`* parameter is not empty, its contents are searched for *.xml files (FieldList files) whose contents appear as the Data Location tables.

Additional `FieldList` files may be manually added to the *`Discovery Path`* directory to be found during discovery. Alternatively, a custom *`Discovery Path`* directory may be established whose contents may be completely independent of the Adapter distribution.

In all cases, the result of a successful discover gesture from which the user may provision a source stream in the model. Once the source stream is instantiated, the user must manually make the symbol a key field.

The *`Map File`* is a `configFilename`, making it directly editable by the Aleri Studio.

The FieldList file may contain any combination of valid FIDs and PseudoFields. See Aleri Activ documentation for more information. PseudoFields include:

- _eventtype
- _hirestimestamp
- _image (L2 only)
- _item, _itemname or _symbol
- _orderid
- _sequencenumber
- _stale
- _trend

- _updatenumber

Activ connectors still require a MAP file, which may reference additional parameters but no other streams.

The advanced runtime parameters are used for remote execution. These parameters are paths in the remote machine's syntax.

For information on how to test your plug-in connector and details about these connectors, see *Guide to Programming Interfaces* .

Parameters (Basic)

`Installation Path`	Absolute path to the Adapter's installation directory. It must have the same value as used by `ALERI_ACTIV_HOME` Type: directory Use: Required Default: $ALERI_ACTIV_HOME
`Map File`	path to map file Maps the data from the vendor's format to the Sybase Aleri Streaming Platform format. This parameter is necessary for connectors that do not have Data Discovery. Mapping specifies what data is of interest and how it will be placed in a source stream of a data model. This is referred to as a Map file in Aleri Activ Adapter documentation. Type: configFilename Use: Required Default: aleri/adapters/activ/
`Discovery Path`	path to the Adapter discovery directory Type: directory Use: Optional (required for discovery)
`ActivUser`	This parameter is a part of the necessary credentials for the ActivFinancial Content Gateway. Type: string Use: Required Default: None
`ActivPassword`	This parameter is a part of the necessary credentials for the ActivFinancial Content Gateway. Type: password Use: Required

	Default: None
`User`	The user name for the Sybase Aleri Streaming Platform. You can set this to match your authentication method. Type: string Use: Optional (may be skipped if the authentication method is set to none) Default: None
`Password`	The password for the Sybase Aleri Streaming Platform. You can set this to match your authentication method. Type: password Use: Optional Default: None

Parameters (Advanced)

`Directory (runtime)`	runtime path to Adapter installation Type: string Use: advanced
`Discovered Table`	name of discovered table; filled in by the Aleri Studio Type:tables Use:advanced
`Map File (runtime)`	The parameter is the runtime path to map file. If this parameter is empty, the connector uses the basic parameter, *`Map File`* . Type: string Use: advanced Default: This field should be left blank.

Known Limitations:

- You must perform the following steps in order to use this connector with discovery:

  1. Convert the `ActivFinancialTableTemplateSpreadsheet.xls` you received in your Activ SDK to a .csv file.
  2. You must then save the `TableTemplateSpreadsheet.csv` in `$ALERI_ACTIV_HOME/config.`

  If you don't perform the preceding steps and the `.csv` file isn't found, you may see the following error message:

```
Exception in thread "main" java.io.FileNotFoundException: ...\config\TableTemp
lateSpreadsheet.csv (The system cannot find the file specified)
```

- When the Sybase Aleri Streaming Platform is started by the Aleri Studio, connectors are started by the Sybase Aleri Streaming Platform engine. If an external Adapter is being started by a connector, it must reside on the same machine as the Sybase Aleri Streaming Platform engine. This configuration is seen for example when the Aleri Studio is running on Windows, with Remote Execution of the Aleri CEP engine on a UNIX machine.
- The configuration of Activ-facing portion of the Aleri Activ Adapter cannot be done from within the Aleri Studio. It requires manual editing of the Adapter's `.ini` file.
- Inbound Plug-in connectors only deal with exactly one source stream.
- You must manually configure an external Adapter rather than a connector to use complex features such as a Finalizer.

F.2. Sybase Aleri Streaming Platform Input

It receives data from a stream in another or the same instance of the Sybase Aleri Streaming Platform. If the connection becomes broken, the connector tries to re-establish it. The connector can be used to create complex composite models. It supports discovery.

Parameters

`Server`	Server host name of the other instance. Use: Required
`Port`	Server control port, or -1 to read from the Ephemeral Port File. Type: int Use: Required
`Ephemeral Port File`	File that will contain the server port number, if port is -1. Use: Advanced
`Use SSL`	Whether to use the SSL encryption wrapping. Type: boolean Use: Optional Default: False
`Authentication`	Authentication mechanism to use for remote Sybase Aleri Streaming Platform instance. Type: choice of None, PAM, RSA, or Kerberos™ V5 Use: Required

`User`	User ID for the platform connection Use: Optional (may be skipped if authentication set to none)
`Password`	Password for PAM authentication Use: Optional
`RSA Key File`	RSA private key file name and location, for RSA authentication. Use: Optional Type: filename
`RSA Key File (Runtime)`	RSA private key file at run time, if different from discovery time. Use: Optional Type: string
`Remote Name Tag`	Symbolic name under which this connection will show on the remote platform. Use: Advanced
`Remote Stream`	Stream to subscribe to on another Platform instance. Use: Advanced
`Include Base Content`	Start by receiving the initial contents of the stream, not just the updates. Type: boolean Use: Optional Default: False
`Lossy Subscription`	If the reader can not keep up, individual updates may be dropped. Type: boolean Use: Advanced Default: False
`Drop Connection If Can Not Keep Up`	If the reader can not keep up, the connection will be dropped and attempt to reconnect. Type: boolean Use: Advanced Default: False
`Convert to Safe Opcodes`	It converts the opcodes INSERT and UPDATE to UPSERT, DELETE to SAFEDELETE. Type: boolean

	Use: Advanced Default: False
`Skip Deletes`	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
`Pulse Period (seconds)`	Non-zero value enables the pulsed updates with this period. Use: Advanced Default: 0
`Maximum Buffer Size`	It is the maximum number of records to queue up before dropping a connection if a subscription cannot keep up and becomes marked droppable. Type: Unsigned Integer Use: Advanced Default: 8000
`Retry Period (seconds)`	Period for trying to re-establish an outgoing connection, in seconds, or 0 for a one-time attempt. Type: uint Use: Advanced Default: 1
`Enter Initial State`	When the connector enters the initial loading state. Use: Advanced Default: auto
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced
`Base Drain Timeout (milliseconds)`	Maximum time (in milliseconds) to receive all base data for a connected stream before the connected Sybase Aleri Streaming Platform forces a disconnect. Type: uint Use: Advanced Default: 8000

Known Limitations:

- Be careful if you create loops in the data flow.

F.3. Sybase Aleri Streaming Platform Output

Send data to a source stream in another, or the same, instance of the Sybase Aleri Streaming Platform. If the connection becomes broken, the connector tries to re-establish it. Can be used to create complex composite models.

This connector can now be configured to send only the base state of the stream. It sends the data once and exits, but it can be restarted later.

Parameters

`Server`	Server host name of the other instance. Use: Required
`Port`	Server control port, or -1 to read from the Ephemeral Port File. Type: int Use: Required
`Ephemeral Port File`	File that will contain the server port number, if port is -1. Use: Advanced
`Use SSL`	Whether to use the SSL encryption wrapping. Type: boolean Use: Optional Default: False
`Authentication`	Authentication mechanism to use for remote Sybase Aleri Streaming Platform instance. Type: choice of None, PAM, RSA, or Kerberos V5 Use: Required
`User`	User ID for the platform connection Use: Optional (may be skipped if authentication is set to none)
`Password`	Password for PAM authentication Use: Optional
`RSA Key File`	RSA private key file name and location, for RSA authentication. Use: Optional

	Type: filename
`RSA Key File (Runtime)`	RSA private key file at run time, if different from discovery time. Use: Optional Type: string
`Retry Period (seconds)`	Period for trying to re-establish an outgoing connection, in seconds. Type: uint Use: Advanced Default: 1
`Remote Stream`	Stream to publish to on another Platform instance. Use: Advanced
`Include Base Content`	Start by recording the initial contents of the stream, not just the updates. Type: boolean Use: Optional Default: False
`Only Base Content`	It only sends once the initial contents of the stream. Type: boolean Use: Advanced Default: False
`Remote Name Tag`	Symbolic name under which this connection will show on the remote platform. Use: Advanced
`Confirm Receipt`	Check the publishing confirmations from the remote platform for success. Type: boolean Use: Advanced Default: False
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- Be careful if you create loops in the data flow.

F.4. Bloomberg Plug-in

The Bloomberg Plug-in connects to the Bloomberg ServerAPI to receive Bloomberg market data. It can be configured on any source stream as an inbound data location. The authentication method is set to that of the Sybase Aleri Streaming Platform: none, pam, rsa, or gssapi.

The Bloomberg Plug-in connector requires Aleri Bloomberg Adapter version 2.1 or later to be installed. You can refer to the Aleri Bloomberg Adapter documentation for details about its installation and configuration. Please contact your Sybase sales representative for more information about the Adapter.

Parameters

`Connector Directory Path`	Specify the absolute path to the Adapter's installation directory. This parameter is ignored if the *`Connector Remote Directory Path`* parameter is supplied. Type: directory Use: Required Default: None
`Configuration File Path`	Specify the absolute path to the Adapter's configuration file. This parameter is ignored if the *`Remote Configuration File Path`* parameter is supplied. Type: configFilename Use: Required Default: None
`Discovery Directory Path`	Specify the absolute path to the Adapter's discovery directory. Type: directory Use: Required Default: None
`Connector Remote Directory Path`	Specify the path to the connector remote base directory (for remote execution only). If this parameter is supplied, the *`Connector Directory Path`* parameter is ignored. Type: string Use: Advanced Default: None
`Remote Configuration File Path`	Specify the path to the connector's remote configuration file (for remote execution only). If this parameter is supplied, the *`Configuration File Path`* parameter is ignored. Type: string

Use: Advanced

Default: None

F.5. Configuring Coral8 Inbound and Outbound Connectors

Coral8 Inbound and Outbound connectors let you establish incoming or outgoing data links with streams in a running Coral8 project. However the connectors require special processing to convert data in Coral8 to a format that can be processed in the Sybase Aleri Streaming Platform and vice versa. These connectors are only available once the Coral8 connector overlay package (on the Sybase Download web site) is installed on top of the Sybase Aleri Streaming Platform Release 3.1.4 or higher.

F.5.1. Data Types

Connectors map the following data types as described below.

C8_INT	int32
C8_LONG	int64
C8_FLOAT	double
C8_STRING	string
C8_TIMESTAMP	timestamp or date. Coral8 timestamps are in microseconds. Connectors perform the appropriate data conversion when reading or writing to a *C8_TIMESTAMP* field.
C8_FLOAT/ C8_LONG	money. The Sybase Aleri Streaming Platform has a money data type. This can be associated with either a float or a long on the Coral8 side. Depending on what it is mapped to, money is converted using the money factor currently in effect in the Sybase Aleri Streaming Platform.

ALERI_OPS and *C8_TIMESTAMP* are special fields, and these names are reserved for use by the connectors.

F.5.2. Coral8 Timestamps

Coral8 messages include an implicit timestamp. The corresponding stream in the Sybase Aleri Streaming Platform may define a field name C8_TIMESTAMP in order to handle it. If defined and the connection parameter *Handle C8 Timestamp* is true, the connectors use the field to correspond to the Coral8 message timestamp.

F.5.3. Operations

The Sybase Aleri Streaming Platform supports insert, delete, update, and upsert operations. You can have a field named ALERI_OPS in the stream to handle these operation codes. If this is defined and the connection parameter *Handle Aleri Operation Codes* is true, the connectors populate the field when writing to and use the field when reading from Coral8.

F.6. Coral8 Inbound

Parameters

`Coral8 host`	the name of the machine where the Coral8 server is running Type: string Use: Required Default: None (required parameter)
`Coral8 server port`	the port number on which the Coral8 server is listening for incoming connections Type: string Use: Required Default: None (required parameter)
`Coral8 workspace`	the name of the workspace containing the stream to connect to Use: Required Default: None (required parameter)
`Coral8 project`	the name of the project containing the stream to connect to Type: string Use: Required Default: None (required parameter)
`Coral8 Stream`	the name of the output stream from which to read data Type: string Use: Required Default: None (required parameter)
`Handle C8 Timestamp`	If set to true, and a field named C8_TIMESTAMP is defined in the target Sybase Aleri Streaming Platform stream, the input connector fills in this field with the value of the Coral8 message timestamp. Type: boolean Use: Advanced Default: True
`Handle Aleri Operation Codes`	If set to true, and a field named ALERI_OPS exists in the source Coral8 output stream schema, the value of that field is used to determine the type of message each data row generates. Recognized opcodes are `i` for insert, `d` for delete, `u` for update, and `U` for upsert. Type: boolean Use: Advanced

	Default: True
`Debug`	If set to true, the connector outputs debugging messages. Type: boolean Use: Advanced Default: False

F.7. Coral8 Outbound

Parameters

`Coral8 host`	the name of the machine where the Coral8 server is running Type: string Use: Required Default: None (required parameter)
`Coral8 server port`	the port number on which the Coral8 server is listening for incoming connections Type: string Use: Required Default: None (required parameter)
`Coral8 workspace`	the name of the workspace containing the stream to connect to Use: Required Default: None (required parameter)
`Coral8 project`	the name of the project containing the stream to connect to Type: string Use: Required Default: None (required parameter)
`Coral8 Stream`	the name of the input stream to which to write data Type: string Use: Required Default: None (required parameter)
`Handle C8 Timestamp`	If set to true, and a field named C8_TIMESTAMP is defined in the target Sybase Aleri Streaming Platform stream, the input connector fills in this field with the value of the Coral8 message

	timestamp. Type: boolean Use: Advanced Default: True
`Handle Aleri Operation Codes`	If set to true, and a field named ALERI_OPS exists in the source Coral8 output stream schema, the value of that field is used to determine the type of message each data row generates. Recognized opcodes are `i` for insert, `d` for delete, `u` for update, and `U` for upsert. Type: boolean Use: Advanced Default: True
`Debug`	If set to true, the connector outputs debugging messages. Type: boolean Use: Advanced Default: False

F.8. Database Input

Receives data from a database table. May be used to poll the table periodically and receive the updates. The exact required parameters depend on the type of RDBMS. If the parameter "SQL Query" is specified, it allows you to override the table selection and get the data from an arbitrary query.

This connector supports discovery.

Parameters

`Database Type`	The brand of RDBMS server. The choices are Oracle®, DB2®, kdb+, PostgreSQL or Microsoft SQL Server®. Sybase ASE, Netezza® and Teradata® databases are also options, but users must obtain the driver from the vendor to install for the Sybase Aleri Streaming Platform. Refer to *the Administrator's Guide* for more information. Use: Required
`Server`	Name or IP address of the database server machine. Use: Required
`Port`	IP port of the database listener Type: int Use: Required

Instance	Instance Name of database instance Use: Optional
Database	Name of database Use: Optional
User	User ID for the database connection Use: Required
Password	Password for the database connection Use: Optional
Input table name (runtime)	The name of the table to select data from Use: Advanced
SQL Query (runtime)	This parameter is used only when the Input is not specified. It defines an arbitrary SQL query that is executed against the database to generate a result set whose records are used as input to the Sybase Aleri Streaming Platform. If this parameter is not defined, the implicit query "SELECT * FROM <Input table name> is used. Use: Advanced
Poll Period (seconds)	Period for polling for new contents, in seconds. Unsigned Integer Use: Advanced
Convert to Safe Opcodes	It converts the opcodes INSERT and UPDATE to UPSERT, DELETE to SAFEDELETE. Type: boolean Use: Advanced Default: False
Skip Deletes	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
Date Format	Format string to parse date values Type: string Use: Advanced

	Default: %Y-%m-%d %H:%M:%S
`Timestamp Format`	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%d %H:%M:%S
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- When polling this must be the only connector.
- Any data updates received from any other source will be undone on the next poll.
- If the connector is for Sybase ASE, the connection is always made to the default database even if you specify another database.

F.9. Database Output

Sends data to a database table. The table may be truncated when the connector starts. The exact required parameters depend on the type of RDBMS.

Database output connectors have the following rules for Timestamp and Date columns in the Sybase Aleri Streaming Platform:

- ISO format= %Y-%m-%d %H:%M:%S", for example, "1964-04-01 17:12:00
- Sybase Aleri Streaming Platform Timestamp columns get ISO formating and wrapped using the JD-BC® timestamp escape:

  ```
              {ts  '<ISOso formatted timestamp>'}
  ```

  when placed into a SQL insert, update or delete statement. You don't have to define the formatting for timestamps.
- Sybase Aleri Streaming Platform Date columns get formatted to the one specified in setting the date-Format connector parameter, with a default of ISO format, and wrapped using simple single quote characters, such as '<iso or user-formatted date>' when placed into a SQL insert, update or delete statement.

A common example of specifying a different dateFormat is when inserting a Sybase Aleri Streaming Platform Date column into an Oracle Date column. The default Oracle date format is: "04-Apr-1964 17:12:00", so you would specify that the dateFormat parameter is "d-%b-%Y %H:%M:%S."

Parameters

`Database Type`	The brand of RDBMS server. The choices are Oracle, DB2, kdb+, Microsoft SQL Server® or PostgreSQL. Sybase ASE, Netezza and Teradata databases are also options, but users must obtain the driver from the vendor to install for the Sybase Aleri Streaming Platform. Refer to *the Administrator's Guide* for more information. Use: Required
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced
`batchLimit`	If this parameter is 1, its behavior remains with SQL insert, update and delete statements being performed one at a time. But if you set `batchLimit` to a value greater than one, such as 1024, then the JDBC batch mechanism is used, which greatly increases the performance when writing to the database. Type: uint Use: Advanced Default: 1
`kdb+ schema mapping`	If set, the output to the kdb database is by the native kdb "q" interface instead of the "SQL" JDBC interface, allowing for more complete and efficient handling of most kdb data types. To trigger the "q" mode for kdb+ (JDBC) output, make sure you have the "exportMap" parameter specified, which is labeled "kdb+ schema mapping" in the advanced tab, and set it for one character/database column. The valid characters are:

Format	**kdb+ Type**
h	short
i	int
l	long
e	real (single)
f	float (double)
c	char (1 character)
C	list-of-char
s	symbol
d	date
z	datetime
t	time (single)

Type: string

Use: Optional

`Server`	Name or IP address of the database server machine.

	Use: Required
Port	IP port of the database listener Type: int Use: Required
Instance	Instance Name of database instance Use: Optional
Database	Name of database Use: Optional
User	User ID for the database connection Use: Required
Password	Password for the database connection Use: Optional
Output table name (runtime)	The name of the table which to push data. Use: Advanced
Date Format	Format string to parse date values Type: string Use: Advanced Default: %Y-%m-%d %H:%M:%S
Timestamp Format	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%d %H:%M:%S
Include Base Content	Outputs initial stream contents in addition to stream updates. Type: boolean Use: Advanced Default: False
Only Base Content	Send only the initial contents of the stream Type: boolean Use: Advanced

	Default: False
`Truncate database table`	Start by truncating the database table, then populating with streaming data. Type: boolean Use: Advanced Default: False

Known Limitations:

- The table must already exist.
- Each row translates to an SQL statement so updates are reasonably slow.

F.10. File CSV Input

Reads a file in Aleri's delimited format. Can be used to poll for new data being appended to the file. The file may be without the header (same as accepted by **sp_convert**), or with header specifying the field names. This connector does support discovery.

Parameters

`Delimiter`	Symbol used to separate the columns. Use: Advanced Default: Comma `(,)`
`Has Header`	Whether the first line of the file contains the description of the fields. Type: boolean Use: Advanced Default: False
`Directory`	Location of the data files. Use: Required
`Directory (runtime)`	Location of the data files at run time, if different from discovery time. Use: Advanced
`File Pattern`	Pattern to look up files for discovery Use: Advanced Default: *.csv

File (in Directory)	File to read Use: Advanced
Poll Period (seconds)	Period for polling for new contents, in seconds. Type: uint Use: Advanced
Convert to Safe Opcodes	It converts the opcodes INSERT and UPDATE to UPSERT, and converts DELETE to SAFEDELETE. Type: boolean Use: Advanced Default: False
Skip Deletes	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
Date Format	Format string for parsing date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Format string for parsing timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
expectStreamNameOpcode	If true, the first two fields are interpreted as stream name and Aleri op code respectively. Messages with unmatched stream names are discarded. Type: boolean Use: Optional Default: False
Block Size	Number of records to block into one pseudo-transaction Type: int Use: Advanced Default: 1

`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- When polling, data may only be appended to the file, but the file may not be overwritten or replaced. The stream name in the file rows is ignored, all the data is sent to the same stream.
- For discovery to work correctly, make sure to set the delimiter character and header presence flag to match the actual data.
- You should not mix the files with different delimiters or with/without headers in the same directory or files with wrong delimiters or headers won't be correctly discovered.

F.11. File CSV Output

Write data as a file in Aleri's delimited format. The file may be without the header (same as accepted by **sp_convert**), or with header specifying the field name.

Parameters

`Delimiter`	Symbol used to separate the columns. Use: Advanced Default: Comma `(,)`
`Has Header`	Whether the first line of the file contains the description of the fields. Type: boolean Use: Advanced Default: False
`Directory`	Location of the data files. Use: Required
`Directory (runtime)`	Location of the data files at run time, if different from discovery time. Use: Advanced
`File Pattern`	Pattern to look up files for discovery Use: Advanced Default: *.csv
`File (in Directory)`	File to write Use: Advanced

`Include Base Content`	Starts by recording the initial contents of the stream, not just the updates. Type: boolean Use: Optional Default: False
`Only Base Content`	Send only the snapshot of initial contents of the stream, once. Type: boolean Use: Optional Default: False
`Date Format`	Format string to parse date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Prepend StreamNameOpcode`	If true, each message will start with the stream name and the Aleri op code. Type: boolean Use: Optional Default: False
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

Data discovery is not supported.

F.12. File FIX Input

Reads FIX messages from a file and writes them as stream records. Each stream hosts FIX messages of a certain type. Messages of any other FIX type are ignored. All FIX fields except the following are being written in the same order in stream columns:

- BeginString

- BodyLength
- MsgType
- CheckSum

The names of the stream columns must correspond to the FIX protocol specification.

Parameters

`FIX Version`	Version of the FIX protocol. Type: choice Use: Required Default: 4.2
`FIX Message Type`	The type of messages hosted by the stream Type: string Use: Required Default: None (required parameter)
`File`	Path to the input file Type: string Use: Required Default: None (required parameter)
`Date Format`	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- **This connector is not a full FIX Engine**. If you require a full FIX Engine, please contact a Sybase sales representative for information about the Aleri FIX Engine Adapter version 1.0.
- Supports only FIX versions 4.2 and 4.3.

- Repeating groups and components are not supported.
- Only supports insert Opcode.

F.13. File FIX Output

Writes stream data as FIX messages to a file. Each stream hosts FIX messages of a certain type. Messages are written to file contiguously, with no line feeds. The following FIX fields are generated by the connector:

- BeginString
- BodyLength
- MsgType
- CheckSum

The rest of the fields must be written in appropriate order in stream columns. The names of the stream columns must correspond to the FIX protocol specification.

Parameters

`FIX Version`	Version of FIX Protocol Type: choice Use: Required Default: 4.2
`FIX Message Type`	The type of messages hosted by the stream Type: string Use: Required Default: None (required parameter)
`File`	Path to the output file Type: string Use: Required Default: None (required parameter)
`Date Format`	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format

Type: string

Use: Advanced

Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- **This connector is not a full FIX Engine**. If you require a full FIX Engine, please contact a Sybase sales representative for information about the Aleri FIX Engine Adapter version 1.0.
- Only versions 4.2 and 4.3 of FIX are supported.
- Repeating groups and components are not supported.
- Data discovery is not supported.
- Only supports insert Opcode.

F.14. File XML Input

It reads a file in AleriML format. This connector can be used to poll for new data being appended to the file. The File XML Input connector supports discovery.

Parameters

`Directory`	Location of the data files. Use: Required
`Directory (runtime)`	Location of the data files at run time, if different from discovery time. Use: Advanced
`File Pattern`	Pattern to look up files for discovery Use: Advanced Default: *.xml
`File (in Directory)`	File to read Use: Advanced
`Poll Period (seconds)`	Period for polling for new contents, in seconds. Type: uint Use: Advanced
`Convert to Safe Opcodes`	It converts the opcodes INSERT and UPDATE to UPSERT, and converts DELETE to SAFEDELETE. Type: boolean

	Use: Advanced Default: False
`Skip Deletes`	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
`Date Format`	Format string for parsing date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string for parsing timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`matchStreamName`	If true, the XML element name will be matched against the stream name. Unmatched messages will be discarded. Type: boolean Use: Optional Default: False
`Block Size`	Number of records to block into one pseudo-transaction Type: int Use: Advanced Default: 1
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- When polling, data may only be appended to the file; the file may not be overwritten or replaced.
- The stream name in the file entries is ignored.
- Don't mix the data files and model XML files in the same directory or the Sybase Aleri Streaming Platform XML files will be discovered as invalid.

F.15. File XML Output

It writes data as a file in the AleriML format.

Parameters

`Directory`	Location of the data files. Use: Required
`Directory (runtime)`	Location of the data files at run time, if different from discovery time. Use: Advanced
`File Pattern`	Pattern to look up files for discovery. Use: Advanced Default: *.xml
`File (in Directory)`	File to read. Use: Advanced
`Include Base Content`	Start by recording the initial contents of the stream, not just the updates. Type: boolean Use: Optional Default: False
`Only Base Content`	Send only the snapshot of initial contents of the stream, once. Type: boolean Use: Optional Default: False
`Date Format`	Format to parse the date values. Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format to parse the timestamp values. Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Field Mapping`	Mapping between the in-platform and external fields Type: string

Use: Advanced

F.16. FIX Plug-in

The FIX Plug-in Adapter runs a full FIX engine as a separate process. It uses any number of file, socket and session connectors to send and receive FIX messages. It can be configured on any source stream as an inbound data location. The authentication method is set to that of the Sybase Aleri Streaming Platform: none, pam, rsa, or gssapi.

The standalone Aleri FIX connector must be installed to use this Plug-in. Please contact a Sybase sales representative for information about the standalone Aleri FIX connector version 1.0. Refer to the Aleri FIX connector documentation for details about its installation and configuration.

Parameters

`Connector Directory Path`	Specify the absolute path to the standalone connector installation directory. This parameter is ignored if the *`Connector Remote Directory Path`* parameter is supplied. Type: directory Use: Required Default: None
`Configuration File Path`	Specify the absolute path to the Adapter's configuration file. This parameter is ignored if the *`Remote Configuration File Path`* parameter is supplied. Type: configFile Use: Required Default: None
`Connector Remote Directory Path`	Specify the path to the connector remote base directory (for remote execution only). If this parameter is supplied, the *`Connector Directory Path`* parameter is ignored. Type: string Use: Advanced Default: None
`Remote Configuration File Path`	Specify the path to the connector's remote configuration file (for remote execution only). If this parameter is supplied, the *`Configuration File Path`* parameter is ignored. Type: string Use: Advanced Default: None

F.17. HTTP Plug-in

The HTTP Plug-in connector receives SQL queries wrapped in HTTP requests from a client application, such as a Web browser and sends chunk-coded stream content back to the client. The plug-in can be configured on any source stream as an inbound data location. The authentication method is set to that of the Sybase Aleri Streaming Platform: none, pam, rsa, or gssapi.

The Aleri HTTP Adapter version 1.0 or later must be installed to use this Plug-in. Please contact your Sybase sales representative for more information about the Adapter.

Parameters

`Connector Directory Path`	Specify the absolute path to the Adapter's installation directory. This parameter is ignored if the *`Connector Remote Directory Path`* parameter is supplied. Type: directory Use: Required Default: None
`Configuration File Path`	Specify the absolute path to the Adapter's configuration file. This parameter is ignored if the *`Remote Configuration File Path`* parameter is supplied. Type: configFilename Use: Required Default: None
`Connector Remote Directory Path`	Specify the path to the connector remote base directory (for remote execution only). If this parameter is supplied, the *`Connector Directory Path`* parameter is ignored. Type: string Use: Advanced Default: None
`Remote Configuration File Path`	Specify the path to the connector's remote configuration file (for remote execution only). If this parameter is supplied, the *`Configuration File Path`* parameter is ignored. Type: string Use: Advanced Default: None

F.18. IDC Plug-in

The IDC (Interactive Data Corporation) Plug-in connects to an IDC PlusFeed or PlusBook source to receive Level 1 and Level 2 market data. It can be configured on any source stream as an inbound data location. The authentication method is set to that of the Sybase Aleri Streaming Platform: none, pam, rsa, or gssapi. This connector supports discovery.

The Aleri IDC Adapter version 1.0 or later must be installed to use this Plug-in. You can refer to the Aleri IDC Adapter documentation for details about its installation and configuration. Please contact your Sybase sales representative for more information about the Adapter.

Parameters

`Connector Directory Path`	Specify the absolute path to the Adapter's installation directory. This parameter is ignored if the *`Connector Remote Directory`* Path parameter is supplied. Type: directory Use: Required Default: None
`Configuration File Path`	Specify the absolute path to the Adapter's configuration file. This parameter is ignored if the *`Remote Configuration File Path`* parameter is supplied. Type: configFile Use: Required Default: None
`Discovery Directory Path`	Specify the absolute path to the Adapter's discovery directory. Type: directory Use: Required Default: None
`Connector Remote Directory Path`	Specify the path to the connector remote base directory (for remote execution only). If this parameter is supplied, the *`Connector Directory Path`* parameter is ignored. Type: string Use: Advanced Default: None
`Remote Configuration File Path`	Specify the path to the connector's remote configuration file (for remote execution only). If this parameter is supplied, the *`Configuration File Path`* parameter is ignored. Type: string Use: Advanced Default: None

F.19. JMS CSV Input

Subscribes and reads text messages formatted as a delimited list of values from JMS queue or topic, then writes the messages as stream records. If opted, the first two fields in every message are interpreted as the stream name and Aleri op code respectively. An empty string is a valid value. This connector supports discovery.

Parameters

delimiter

Specifies the delimiter between fields. The default is a comma.

Type: string

Use: Required

Default: Comma

Connection Factory

The Java class of the connection factory used to connect to the message broker. Valid values are:

- org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ)
- com.sun.messaging.ConnectionFactory (Glassfish OpenMQ)
- com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS)

  To use this Java class, you must first obtain the IBM Websphere® MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details.
- oracle.jms.AQjmsConnectionFactory (Oracle AQ)

Type: string

Use: Required

Default: None

Host Name

Host name or IP address of the message broker

Type: string

Use: Required

Default: None (required parameter)

port

Connection port of the message broker

Type: uint

Use: Required

Default: None (required parameter)

imqConsumerFlowLimit

It is specific to Glassfish OpenMQ. The upper limit of the number of messages per consumer that will be delivered and buffered in the MQ client.

Type: string

	Use: Advanced Default: 10
Destination Type	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
Destination Name	Name of the destination (queue or topic). Type: string Use: Required Default: None (required parameter)
Stream Name Opcode Expected	If true, the first two fields are interpreted as stream name and Aleri op code respectively. Messages with unmatched stream names are discarded. Type: boolean Use: Advanced Default: False
Date Format	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Timestamp format Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitation:

No reconnection attempt is made when the connection to the message broker is lost.

F.20. JMS CSV Output

Publishes stream data as text messages formatted as delimited list of values to a JMS queue or topic. If opted, each message will be prepended with the stream name and the Aleri op code. If a column has a null value, an empty string will be added to the list.

Parameters

`delimiter`	Delimiter between fields; the default delimiter is a comma. Type: string Use: Required Default: Comma
`Connection Factory`	The Java class of the connection factory used to connect to the message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ) • com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS) To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details. • oracle.jms.AQjmsConnectionFactory (Oracle AQ) Type: string Use: Required Default: None
`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required

Default: None (required parameter)

Delivery Mode

The delivery mode has valid values of PERSISTENT and NON_PERSISTENT.

Type: choice

Use: Optional

Default: PERSISTENT

Column TO Property Map

A Comma-delimited list of ColumnName=PropertyName mappings. For each mapped column name, the outbound message will contain a corresponding JMS property whose value equals the column value.

The following is an example:

```
MyColumn1=MyProperty1,MyColumn2=MyProperty2
```

No spaces are accepted within the value of this parameter. Setting the JMS properties enables the message filtering on the message broker side using the JMS selector mechanism.

Type: string

Use: Advanced

Default: None

Prepend Stream Name, Opcode

If true, each message will start with the stream name and the Aleri op code.

Type: boolean

Use: Advanced

Default: False

Date Format

Date format

Type: string

Use: Advanced

Default: %Y-%m-%dT%H:%M:%S

Timestamp Format

Timestamp format

Type: string

Use: Advanced

Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- No reconnection attempt is made when the connection to the message broker is lost.

F.21. JMS Custom Input

Subscribes and reads custom-formatted Java object messages from a JMS queue or topic, then writes the messages as stream records. The format conversion is performed by a custom-provided implementation of the following interface:

```
package com.aleri.connectors;

public interface ExternalToAleriConverter {

    public AleriMessage externalToAleri(Serializable externalMessage) throws Exception;
}
```

The objects returned by the externalToAleri method should implement the following interface:

```
package com.aleri.connectors;

public interface AleriMessage extends Serializable {

    public String getStreamName();

    public String getOpCode();

    public Map<String, Serializable> getColumnValues();
}
```

The objects returned by the `getStreamName`, `getOpCode`, `getColumnValues` methods are interpreted, respectively, as the name of the stream to write to, the op code, and the stream record as a column name to value map. The Java classes of column values must correspond to column types as shown in Section F.25, “JMS Object Array Input”.

Implementations of the ExternalToAleriConverter interface must provide a constructor with a single argument of java.lang.String type or alternatively, a default constructor with no arguments.

Records with unmatched stream names are ignored. Records with null op code are interpreted as upserts. The values of non-key columns may be absent or null.

It is the user's responsibility to provide a JAR archive containing an implementation of the ExternalToAleriConverter interface. The archive should be placed in the *lib* subfolder of the Sybase Aleri Streaming Platform installation folder.

If an implementation is not provided, the default implementation is used, whereby it is assumed that the external message is an instance of the DefaultAleriMessage class and no actual conversion is performed.

This connector supports discovery.

Parameters

Connection Factory	The Java class of the connection factory used to connect to the message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ)

	• com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS) To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details. • oracle.jms.AQjmsConnectionFactory (Oracle AQ) Type: string Use: Required Default: None
`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`imqConsumerFlowLimit`	It is specific to Glassfish OpenMQ. The upper limit of the number of messages per consumer that will be delivered and buffered in the MQ client. Type: string Use: Advanced Default: 10
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required Default: None
`Converter Class Name`	Fully qualified name of a custom-provided implementation of the ExternalToAleriConverter

	Type: string Use: Required Default: ExternalToAleriConverter
Converter Parameter	To be passed as the single argument to the constructor of the custom-provided implementation of the ExternalToAleriConverter interface. Type: string Use: Optional Default: None (the no-arguments constructor will be used).

Known Limitation:

No reconnection attempt is made when the connection to the message broker is lost.

F.22. JMS Custom Output

Publishes stream records as custom-formatted Java objects to a JMS queue or topic. The format conversion is performed by a custom-provided implementation of the following interface:

```
package com.aleri.connectors;

public interface AleriToExternalConverter {

    public Serializable aleriToExternal(AleriMessage aleriMessage) throws Exception;
}
```

See Section F.25, “JMS Object Array Input” for the definition of the AleriMessage interface.

Implementations of the AleriToExternalConverter interface must provide a constructor with a single argument of java.lang.String type or alternatively, a default constructor with no arguments.

The stream name, op code and column name to value map of the aleriMessage object are guaranteed to be valid, even though some non-key column values may be null.

It is the user's responsibility to provide a JAR archive containing an implementation of the AleriToExternalConverter interface. The archive should be placed in the *lib* subfolder of the Sybase Aleri Streaming Platform installation folder.

If an implementation is not provided, the default implementation is used, whereby the aleriMessage object is returned with no actual conversion performed.

Parameters

Connection Factory	The Java class of the connection factory used to connect to the message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ)

	• com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS) To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details. • oracle.jms.AQjmsConnectionFactory (Oracle AQ) Type: string Use: Required Default: None
`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required Default: None
`Delivery Mode`	The delivery mode has valid values of PERSISTENT and NON_PERSISTENT. Type: choice Use: Optional Default: PERSISTENT
`Column to Message Property Map`	A Comma-delimited list of ColumnName=PropertyName mappings. For each mapped column name, the outbound message will contain a corresponding JMS property whose value equals the

	column value. The following is an example: `MyColumn1=MyProperty1,MyColumn2=MyProperty2` No spaces are accepted within the value of this parameter. Setting the JMS properties enables the message filtering on the message broker side using the JMS selector mechanism. Type: string Use: Advanced Default: None
`Converter Class Name`	Fully qualified name of a custom-provided implementation of the AleriToExternalConverter interface. Type: string Use: Required Default: com.aleri.connectors.DefaultAleriToExternalConverter
`Converter Parameter`	To be passed as the single argument to the constructor of the custom-provided implementation of the AleriToExternalConverter interface. Type: string Use: Optional Default: None (the no-arguments constructor will be used).

Known Limitations:

- No reconnection attempt is made when the connection to the message broker is lost.

F.23. JMS FIX Input

Subscribes and reads FIX messages from a JMS queue or topic, then writes the messages as stream records. Each stream hosts FIX messages of a certain type. Messages of any other FIX type are discarded. All FIX fields except the following are stored in the same order in stream columns.

- BeginString

  Subscribes and reads custom-formatted Java object messages from a JMS queue or topic, then writes the messages as stream records. The format conversion is performed by a custom-provided implementation of the following interface:

- BodyLength

- MsgType

- CheckSum

The names of the stream columns must correspond to the FIX protocol specification.

Parameters

`Fix Version`	Version of the FIX protocol Type: choice Use: Required Default: 4.2
`FIX Message Type`	The type of messages hosted by the stream Type: string Use: Required Default: None (required parameter)
`Connection Factory`	The Java class of the connection factory used to connect to the message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ) • com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS) To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details. • oracle.jms.AQjmsConnectionFactory (Oracle AQ) Type: string Use: Required Default: None
`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)

imqConsumerFlowLimit	It is specific to Glassfish OpenMQ. The upper limit of the number of messages per consumer that will be delivered and buffered in the MQ client. Type: string Use: Advanced Default: 10
Destination Type	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
Destination Name	Name of destination (queue or topic) Type: string Use: Required Default: None (required parameter)
Date Format	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Timestamp format Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- **This connector is not a full FIX Engine**. If you require a full FIX Engine, please contact a Sybase sales representative for information about the Aleri FIX Engine Adapter version 1.0.
- Only supports FIX Versions 4.2 and 4.3
- Repeating groups and components are not supported
- No reconnection attempt is made when the connection to the message broker is lost.
- Only supports insert Opcode.

F.24. JMS FIX Output

Publishes stream data as FIX messages to a JMS queue or topic. Each stream hosts FIX messages of a certain type. The following FIX fields are generated by the connector:

- BeginString
- BodyLength
- MsgType
- CheckSum

Parameters

`FIX Version`

Version of the FIX protocol

Type: choice

Use: Required

Default: 4.2

`FIX Message Type`

The type of messages hosted by the stream

Type: string

Use: Required

Default: None (required parameter)

`Connection Factory`

The Java class of the connection factory used to connect to the message broker. Valid values are:

- org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ)
- com.sun.messaging.ConnectionFactory (Glassfish OpenMQ)
- com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS)

  To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details.
- oracle.jms.AQjmsConnectionFactory (Oracle AQ)

Type: string

Use: Required

Default: None

`Host Name`

Host name or IP address of the message broker

Type: string

	Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`imqConsumerFlowLimit`	It is specific to Glassfish OpenMQ. The upper limit of the number of messages per consumer that will be delivered and buffered in the MQ client. Type: string Use: Advanced Default: 10
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required Default: None (required parameter)
`Delivery Mode`	The delivery mode has valid values of PERSISTENT and NON_PERSISTENT. Type: choice Use: Optional Default: PERSISTENT
`Column to Message Property Map`	A Comma-delimited list of ColumnName=PropertyName mappings. For each mapped column name, the outbound message will contain a corresponding JMS property whose value equals the column value. The following is an example: `MyColumn1=MyProperty1,MyColumn2=MyProperty2` No spaces are accepted within the value of this parameter. Setting

	the JMS properties enables the message filtering on the message broker side using the JMS selector mechanism. Type: string Use: Advanced Default: None
`Date Format`	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- **This connector is not a full FIX Engine**. If you require a full FIX Engine, please contact a Sybase sales representative for information about the Aleri FIX Engine Adapter version 1.0.
- Only FIX Versions 4.2 and 4.3 are supported.
- Repeating groups and components are not supported.
- Data discovery is not supported.
- No reconnection attempt is made when the connection to the message broker is lost.
- Only supports insert Opcode.

F.25. JMS Object Array Input

Subscribes and reads messages formatted as arrays of Java objects from a JMS queue or topic, then writes the messages as stream records. If opted, the first two objects in every message are interpreted as stream name and Aleri op code respectively. Null elements in the array will generate null values for the corresponding columns. Column types must correspond to Java classes as follows:

Stream Column Type	**Java Class**
string	java.lang.String
int16	java.lang.Integer
int32	java.lang.Integer
int64	java.lang.Long
money	java.lang.Double

Stream Column Type	**Java Class**
double	java.lang.Double
date	java.util.Date
timestamp	java.util.Date

This connector supports discovery.

Parameters

`Connection Factory`	The Java class of the connection factory used to connect to the message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ) • com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS) To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called "Configuring Non-standard Data Locations" in the *Administrator's Guide* for details. • oracle.jms.AQjmsConnectionFactory (Oracle AQ) Type: string Use: Required Default: None
`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`imqConsumerFlowLimit`	It is specific to Glassfish OpenMQ. The upper limit of the number of messages per consumer that will be delivered and buffered in the MQ client. Type: string Use: Advanced

	Default: 10
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required Default: None (required parameter)
`Expect Stream Name, Op-code`	If true, the first two fields are interpreted as stream name and Aleri op code respectively. Messages with unmatched stream names are discarded. Type: boolean Use: Advanced Default: False

Known Limitation:

No reconnection attempt is made when the connection to the message broker is lost.

F.26. JMS Object Array Output

Publishes stream data as an array of Java objects to a JMS queue or topic. If opted, each message will start with the stream name and the Aleri op code. If a column has a null value, it corresponds to a null element in the array. Column types correspond to Java classes as follows:

Stream Column Type	Java Class
string	java.lang.String
int16	java.lang.Integer
int32	java.lang.Integer
int64	java.lang.Long
money	java.lang.Double
double	java.lang.Double
date	java.util.Date
timestamp	java.util.Date

Parameters

`Connection Factory`	The Java class of the connection factory used to connect to the

	message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ) • com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS) To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details. • oracle.jms.AQjmsConnectionFactory (Oracle AQ) Type: string Use: Required Default: None
`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required Default: None (required parameter)
`Delivery Mode`	The delivery mode has valid values of PERSISTENT and NON_PERSISTENT. Type: choice Use: Optional

	Default: PERSISTENT
Column To Message PropertyMap	A Comma-delimited list of ColumnName=PropertyName mappings. For each mapped column name, the outbound message will contain a corresponding JMS property whose value equals the column value. The following is an example: `MyColumn1=MyProperty1,MyColumn2=MyProperty2` No spaces are accepted within the value of this parameter. Setting the JMS properties enables the message filtering on the message broker side using the JMS selector mechanism. Type: string Use: Advanced Default: None
Prepend Stream Name, Opcode	If true, each message will start with the stream name and the Aleri op code. Type: boolean Use: Advanced Default: False

Known Limitations:

- No reconnection attempt is made when the connection to the message broker is lost.

F.27. JMS XML Input

Subscribes to and reads XML-formatted text messages from a JMS queue or topic, then writes the messages as stream records. Each message must consist of an XML element. If opted, the element name will correspond to the stream name. The ALERI-OPS attribute is optional. If omitted, the message will be interpreted as an upsert. The rest of the attributes must have the same names as the corresponding stream columns. The columns with null values must be omitted. This connector supports discovery.

Parameters

Connection Factory	The Java class of the connection factory used to connect to the message broker. Valid values are: • org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ) • com.sun.messaging.ConnectionFactory (Glassfish OpenMQ) • com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS)

To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details.

- oracle.jms.AQjmsConnectionFactory (Oracle AQ)

Type: string

Use: Required

Default: None

`Host Name`

Host name or IP address of the message broker

Type: string

Use: Required

Default: None (required parameter)

`port`

Connection port of the message broker

Type: uint

Use: Required

Default: None (required parameter)

`imqConsumerFlowLimit`

It is specific to Glassfish OpenMQ. The upper limit of the number of messages per consumer that will be delivered and buffered in the MQ client.

Type: string

Use: Advanced

Default: 10

`Destination Type`

Destination type. Valid values are QUEUE and TOPIC.

Type: choice

Use: Required

Default: QUEUE

`Destination Name`

Name of destination (queue or topic)

Type: string

Use: Required

Default: None (required parameter)

`Match Stream Name`

If true, the XML element name will be matched against the stream name. Unmatched messages will be discarded.

Type: boolean

	Use: Advanced
	Default: False
`Date Format`	Date format
	Type: string
	Use: Advanced
	Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format
	Type:String
	Use: Advanced
	Default: %Y-%m-%dT%H:%M:%S

Known Limitation:

No reconnection attempt is made when the connection to the message broker is lost.

F.28. JMS XML Output

Publishes stream data as XML-formatted text messages to a JMS queue or topic. Each message consists of an XML element whose name is the same as the stream name. The first attribute is the Aleri op code. The rest of the attributes have the same names as the corresponding stream columns. The columns with null values are omitted.

Parameters

`Connection Factory`

The Java class of the connection factory used to connect to the message broker. Valid values are:

- org.apache.activemq.ActiveMQConnectionFactory (ActiveMQ)
- com.sun.messaging.ConnectionFactory (Glassfish OpenMQ)
- com.ibm.mq.jms.MQConnectionFactory (IBM MQ JMS)

  To use this Java class, you must first obtain the IBM Websphere MQ JDBC driver. See the section called “Configuring Non-standard Data Locations” in the *Administrator's Guide* for details.
- oracle.jms.AQjmsConnectionFactory (Oracle AQ)

Type: string

Use: Required

Default: None

`Host Name`	Host name or IP address of the message broker Type: string Use: Required Default: None (required parameter)
`port`	Connection port of the message broker Type: uint Use: Required Default: None (required parameter)
`Destination Type`	Destination type. Valid values are QUEUE and TOPIC. Type: choice Use: Required Default: QUEUE
`Destination Name`	Name of destination (queue or topic) Type: string Use: Required Default: None (required parameter)
`Delivery Mode`	The delivery mode has valid values of PERSISTENT and NON_PERSISTENT. Type: choice Use: Optional Default: PERSISTENT
`Column To Message Property Map`	A Comma-delimited list of ColumnName=PropertyName mappings. For each mapped column name, the outbound message will contain a corresponding JMS property whose value equals the column value. The following is an example: `MyColumn1=MyProperty1,MyColumn2=MyProperty2` No spaces are accepted within the value of this parameter. Setting the JMS properties enables the message filtering on the message broker side using the JMS selector mechanism. Type: string Use: Advanced

	Default: None
`Date Format`	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- No reconnection attempt is made when the connection to the message broker is lost.

F.29. kdb Input Plug-in

The kdb Input Plug-in connector works with the **sp_kdbin** utility to read data from a kdb or a kdb+tick installation into a stream in the Sybase Aleri Streaming Platform. The connector can read both queried and streaming data based on a configuration parameter. This connector supports discovery.

By default, the connector matches the field names (in a case-insensitive manner) to decide the mapping between the source kdb+tick table and the target stream. You also have the option of explicitly specifying the mapping.

Parameters

`Platform User Id`	The user name to connect to the Sybase Aleri Streaming Platform. Type: string Use: Optional Default: None (required parameter)
`Platform Password`	The password associated with the Sybase Aleri Streaming Platform UserID. It is required if using RSA authentication. Type: string Use: Optional Default: None
`RSA Key File`	The full path to the RSA Key private key file. This option must be provided if authentication type is set to RSA.

	Type: string Use: Optional Default: None
`Encrypt`	Whether to encrypt the communication between the connector and Sybase Aleri Streaming Platform. Type: boolean Use: Optional Default: False
`KDB Server`	The name or IP Address of the machine hosting the kdb+tick database. Type: string Use: Required Default: localhost
`KDB Port`	This is the port on which the kdb+tick database is listening. It must be a value between 1 and 65535. Type: int Use: Required Default: 5001
`KDB User`	This is the user id to use to connect to the kdb+tick database. Type: string Use: Optional Default: None
`KDB Password`	This is the password, associated with the user id, to connect to the kdb+tick database. Type: string Use: Optional Default: None
`Source Table/Query`	This specifies either a name of a table in kdb or kdb+tick database or a valid query string, which is used to retrieve the data. You should note that if specifying a query, the plug-in connector retrieves the resulting schema named as 'Query'. You may need to rename any streams generated from the discovered schemas. Type: string Usage: Required

Default: None

`Platform Host Name`	If specified, the plug-in connector uses the specific gateway host and ignores the hostname returned by the Sybase Aleri Streaming Platform. Type: string Use: Optional Default: None
`Field Mapping`	This is an optional parameter that specifies the mapping between the Sybase Aleri Streaming Platform stream column name and the kdb+tick database table column names. The mapping is in the following format. `SPColumn1=KDBColumn1:SpColumn2=KDBColumn2` If this parameter is not provided, the connector will absorb data for only those columns where the target stream column name matches the source table column name in a case-insensitive manner. Use: Optional Default: None
`Streaming Mode`	It determines if the connector should connect to a kdb+tick database and read in streaming data or execute the supplied query and feed the result to the Sybase Aleri Streaming Platform. Allowed values are Stream and Pull. Type: string Use: Required Default: Stream
`Polling Interval`	In non-streaming mode, there is an option to have the supplied query run regularly at a configurable interval. This parameter specifies the polling interval in seconds. A value of 0 indicates that no polling is to be performed - the query is executed just once. Type: int Use: Optional Default: 0
`Block Size`	The maximum number of records that will be sent as a single block to the Sybase Aleri Streaming Platform. The default value is 64. A higher value may increase throughput but will also increase latency. The number of records in the block may be smaller than this value if not enough data is available. Type: int

	Use: Required Default: 64
`Async Mode`	If set to true, the connector will not wait for acknowledgment from the Sybase Aleri Streaming Platform that it received the data. This needs to be set to true in the hot spare configurations to ensure that both the primary and the hot spare has received the data. Type: boolean Use: Optional Default: False
`Use Transaction Blocks`	If set to true, the data will be sent to the Sybase Aleri Streaming Platform as transaction blocks instead of envelopes. It improves performance but will cause all records in the block to be rejected if one of the records in the block fails. Type: boolean Use: Optional Default: False
`Connection Retries`	It allows you to configure the Adapter to connect back to kdb or kdb+tick if the connection breaks during operation. The connector will try to reconnect the configured number of times, waiting for one second before each try. Type: int Use: Optional Default: 1
`Debug`	If set to true, the connector outputs debug messages. Type: boolean Use: Optional Default: False

Known Limitations:

- If the kdb+tick/databases are not running, when the connector tries to make a connection, the connector will wait indefinitely until the kdb+tick database is started. This is only an issue if the kdb+tick database and the Sybase Aleri Streaming Platform are running on different machines.
- If the connection to the database is broken, any updates that happen between the time the connection is broken and re-established are lost.

F.30. kdb Output Plug-in

It is an external connector that streams data from the Sybase Aleri Streaming Platform to a kdb+tick database table.

By default, the connector matches the field names (in a case-insensitive manner) to decide the mapping between the source kdb+tick table and the target stream. You also have the option of explicitly specifying the mapping.

The kdb+tick connector does not currently support data discovery. If you want to use data discovery for tables in the kdb+tick database to create a matching stream in the Sybase Aleri Streaming Platform, use the Database Input connector and then replace it with this connector.

Parameters

`Platform User Id`	user Id for connecting to the Sybase Aleri Streaming Platform
	Type: string
	Use: Optional
`Platform Password`	password for connecting to the Sybase Aleri Streaming Platform
	Type: string
	Use: Optional
`Authentication`	You must select one of the valid options for authentication.
	Choice: None, RSA, PAM and Kerberos V5
	Use: Required
	Default: None
`RSA Key File`	private key file if you use RSA authentication
	Type: string
	Use: Optional
`KDB Server`	host machine running the kdb server
	Type: string
	Use: Required
`KDB Port`	port number which the kdb server is listening on
	Type: int
	Use: Required
`KDB User`	user ID that connects to kdb+tic.
	Type: string
	Use: Optional
`KDB Password`	password that connects to kdb+tic.

	Type: password Use: Optional
Target Table	name of table in kdb+tick where to write data Type: string Use: Required
Source Query	An optional SQL query string to use to retrieve data. If empty, **sp_kdbout** will stream all data from the source stream, which defined it. Type: string Use: Optional
Field Mapping	Explicit field mapping. The mapping is specified as follows: "spfld1=kdbfld1:spfld2=kdbfld2 ..." If mapping is empty, the adapter maps fields by matching their names. Type: string Use: Optional
Streaming Mode	Valid options are stream and push. If mode is stream data, it is sent to kdb+tick using ".u.upd". If mode is push, data is sent using **upsert** command.
Async	If true, **sp_kdbout** sends data to kdb+tick asynchronously. Type: boolean Default: False
BatchSize	If specified, this option sets the maximum number of rows **sp_kdbout** includes in a batch write to kdb+tick. If left empty, **sp_kdbout** uses a value of 5000. Type: integer Default: 5000
Base Data Only	If true, **sp_kdbout** will not stream data existing in the source stream at the time it is started. Type: boolean Default: False
Lossy Subscription	If set to true, **sp_kdbout** will use a lossy subscription to the Sybase Aleri Streaming Platform. This will result in the model dropping data if the adapter cannot keep up. Type: boolean Default: False

`Pulse Interval`	If set to a non-zero value, **sp_kdbout** uses a pulsed subscription. In this mode platform sends the data at intervals, consolidating the data in between. Type: int Default: 0
`Droppable Subscription`	If set to true, the Sybase Aleri Streaming Platform will drop subscription from the adapter, if it cannot keep up with the data. Type: boolean Default: False
`Preserve Transaction Blocks`	If set to true, the subscription from the Sybase Aleri Streaming Platform preserves the transaction boundaries. Type: boolean Default: False
`Debug`	If set to true, **sp_kdbout** will generate debug messages. Type: boolean Default: False
`Ignored Fields`	This is a comma delimited list of kdb field names which **sp_kdbout** will ignore. That means the values for these fields will be ignored and always set to NULL. This parameter can be used to handle kdb datatypes that **sp_kdbout** does not recognize such as list of floats. Type: string Use: Optional
`Omitted Fields`	This is a comma delimited list of kdb field names which **sp_kdbout** will omit from the data message it sends to kdb+tick. It should be used to avoid including fields from the source stream in the Sybase Aleri Streaming Platform which are automatically filled in by kdb+tick. Type: string Use: Optional

F.31. Reuters Marketfeed Inbound Plug-in

The Reuters Marketfeed Inbound Plug-in connector works with the Aleri Reuters Marketfeed Adapter which connects to Reuters Market Data System (RMDS) to receive real-time Level 1 and/or Level 2 market data. A Marketfeed connector can be configured on any source stream as an inbound data location.

This connector is listed as **rmdsMFInPlugin** in the Aleri Studio's Data Location list. It must be used with Aleri Reuters Marketfeed Adapter (binary name=rmds) version 2.2.0 or later. You must use this connector with Aleri Reuters OMM Adapter version 1.2.0 or later. The Adapter, whose executable name is rmds, must be installed per Adapter Guide directions. Plug-in connectors are started on the same machine as the Sybase Aleri Streaming Platform which is controlled by the Remote Execution dialog.

This connector supports discovery.

You can refer to *Aleri Reuters Marketfeed Adapter* documentation for installation and configuration details. Please contact a Sybase sales representative if you are interested in obtaining a Marketfeed Adapter.

The "discover" gesture from the Aleri Studio activates a wrapper script which passes Aleri Studio-editable parameters. If the *`Discover Path`* parameter is not empty, its contents are searched for *.xml files (FieldList files) whose contents appear as the Data Location tables.

Additional *`FieldList`* files may be manually added to the *`Discovery Path`* directory to be found during Discovery. Alternatively, a custom *`Discovery Path`* directory may be established whose contents may be completely independent of the adapter distribution.

In all cases, the result of a successful discover gesture produces Data Location tables from which the user may provision a source stream in the model. Once the source stream is instantiated, the user must manually make the Symbol (RIC) a key field.

The *`Map File`* is a *`configFilename`*, making it directly editable by the Aleri Studio.

The FieldList file may contain any combination of valid FIDs and PseudoFields. PseudoFields are fully documented in the Reuters Adapter Guide and include:

- _hirestimestamp
- _item, _itemname, _symbol or _ric
- _sequenceNumber
- _service, _servicename
- _stale
- _updatenumber

Marketfeed connectors still require a MAP file, which may reference a (Reuters-format) `.cfg` file.

The advanced runtime parameters are used for remote execution. These parameters are paths in the remote machine's syntax.

For information on how to test your plug-in connector and details about these connectors, see *Guide to Programming Interfaces* .

Parameters (Basic)

`Installation Path`	Absolute path to the Adapter's installation directory. It must have the same value as used by `ALERI_REUTERS_HOME`. Type: directory Use: Required Default: $ALERI_REUTERS_HOME

Map File	path to map file Maps the data from the vendor's format to the Sybase Aleri Streaming Platform format. This parameter is necessary for connectors that do not have Data Discovery. Mapping specifies what data is of interest and how it will be placed in a source stream of a data model. This is referred to as a Map file in Aleri Reuters Adapter documentation. Type: configFile Use: Required Default: $ALERI_RMDSMF_HOME/examples/subexample.mf.map.xml
Discovery Path	path to the Adapter discovery directory Type: directory Use: Optional
User	This is the user name for the Sybase Aleri Streaming Platform. You can set this to match your authentication method. Type: string Use: Optional (may be skipped if authentication set to none) Default: None
Password	This is the password for the Sybase Aleri Streaming Platform. You can set this to match your authentication method. Type: password Use: Optional Default: None

Parameters (Advanced)

Directory (runtime)	runtime path to Adapter installation Type:string Use:advanced
Discovered Table	name of discovered table; filled in by the Aleri Studio Type: tables Use: advanced
Map File (runtime)	runtime path to map file Type: string Use: advanced

Default: This field should be left blank.

Known Limitations:

- When the Sybase Aleri Streaming Platform is started by the Aleri Studio, connectors are started by the Sybase Aleri Streaming Platform engine. If an external Adapter is being started by a connector, it must reside on the same machine as the Sybase Aleri Streaming Platform engine. This configuration is seen for example when the Aleri Studio is running on Windows, with Remote Execution of the Aleri CEP engine on a UNIX machine.
- The configuration of Reuters-facing portion of the Aleri Reuters Marketfeed Adapter cannot be done from within the Aleri Studio. It requires manual editing of the Adapter's .cfg file.
- Inbound Plug-in connectors only deal with exactly one source stream.
- You must manually configure an external Adapter rather than a connector to use complex features such as a Finalizer.

F.32. Reuters OMM Inbound Plug-in

The Reuters Open Message Model (OMM) Inbound Plug-in connector works with the Aleri Reuters OMM Adapter, which connects to Reuters Market Data System (RMDS) to receive real-time Level 1 and/or Level 2 market data. The OMM inbound connector can be configured on any source stream as an inbound data location.

This connector is listed as **rmdsOMMInPlugin** in the Aleri Studio's Data Locations list. You must use this connector with Aleri Reuters OMM Adapter version 1.2.0 or later. The Adapter, whose executable name is **rmdsomm**, must be installed per Adapter Guide directions. Plug-in connectors are started on the same machine as the Sybase Aleri Streaming Platform which is controlled by the Remote Execution dialog.

This connector supports discovery.

You can refer to Aleri Reuters OMM Adapter documentation for installation and configuration details. Please contact a Sybase sales representative if you are interested in obtaining an OMM Adapter.

The "discover" gesture from the Aleri Studio activates a wrapper script which passes Aleri Studio-editable parameters. If the *`Discover Path`* parameter is not empty, its contents are searched for *.xml files (FieldList files) whose contents appear as the Data Location tables.

Additional *`FieldList`* files may be manually added to the *`Discovery Path`* directory to be found during discovery. Alternatively, a custom *`Discovery Path`* directory may be established whose contents may be completely independent of the Adapter distribution.

In all cases, the result of a successful discover gesture produces Data Location tables from which the user may provision a source stream in the model. Once the source stream is instantiated, the user must manually make the Symbol (RIC) a key field. For L2 data, a secondary keyfield is also required. The L2 key depends on the message type.

The stream name will be used to establish the OMM message type. The name of the stream is searched for these character patterns which can appear in upper, lower or mixed case. Certain names correlate with specific message types.

Name	Message Type
"mbo", "marketByOrder" or "market_by_order"	MARKET_BY_ORDER

Name	**Message Type**
"mbp", "marketByPrice" or "market_by_price"	MARKET_BY_PRICE
"mp", "marketPrice" or "market_price"	MARKET_PRICE
"mm", "marketmaker" or "market_maker"	MARKET_MAKER

The *`Map File`* is a `configFilename`, making it directly editable by the Aleri Studio.

The FieldList file may contain any combination of valid FIDs and PseudoFields. PseudoFields are fully documented in the *Reuters Marketfeed Adapter Guide* and include:

- _hirestimestamp
- _image
- _item, _itemname, _ric or _symbol
- _marketbyorderkey
- _marketbypricekey
- _marketmakerkey
- _resptypenum
- _sequencenumber
- _stale
- _updatenumber

For Level 2 only:

- _image
- _marketMakerKey
- _marketByOrderKey
- _marketByPriceKey

OMM connectors still require a MAP file, which may reference a (Reuters-format) `.cfg` file.

The advanced runtime parameters are used for remote execution. These parameters are paths in the remote machine's syntax.

For information on how to test your plug-in connector and details about these connectors, see *Guide to Programming Interfaces.*

Parameters (Basic)

`Installation Path`	Absolute path to the Adapter's installation directory. It must have the same value as used by `ALERI_RMDSOMM_HOME`.

	Type: directory Use: Required Default: $ALERI_RMDSOMM_HOME
`Map File`	path to map file Maps the data from the vendor's format to the Sybase Aleri Streaming Platform format. This parameter is necessary for connectors that do not have Data Discovery. Mapping specifies what data is of interest and how it will be placed in a source stream of a data model. This is referred to as a Map file in Aleri Reuters Adapter documentation. Type: filename Use: Required Default: $ALERI_RMDSOMM_HOME/examples/subexample.omm.map.xml
`Discovery Path`	path to the Adapter discovery directory Type: directory Use: Optional
`User`	This is the user name for the Sybase Aleri Streaming Platform. You can set this to match your authentication method. Type: string Use: Optional (may be skipped if authentication set to none) Default: None
`Password`	This is the password for the Sybase Aleri Streaming Platform. You can set this to match your authentication method. Type: password Use: Optional Default: None

Parameters (Advanced)

`Directory (runtime)`	runtime path to Adapter installation Type: string Use: advanced
`Discovered Table`	name of discovered table; filled in by the Aleri Studio Type: tables

	Use: advanced
`Map File (runtime)`	runtime path to map file Type: string Use: advanced Default: This field should be left blank.

Known Limitations:

- When the Sybase Aleri Streaming Platform is started by the Aleri Studio, connectors are started by the Sybase Aleri Streaming Platform engine. If an external Adapter is being started by a connector, it must reside on the same machine as the Sybase Aleri Streaming Platform engine. This configuration is seen for example when the Aleri Studio is running on Windows, with Remote Execution of the Aleri CEP engine on a UNIX machine.
- The configuration of Reuters-facing portion of the Aleri Reuters OMM Adapter cannot be done from within the Aleri Studio. It requires manual editing of the Adapter's .cfg file.
- Inbound Plug-in connectors only deal with exactly one source stream.
- You must manually configure an external Adapter rather than a connector to use complex features such as a Finalizer.

F.33. SMTP Output

Sends an email containing stream records. For each record, the body of the email will contain:

- Stream name
- Column names and values

Parameters

`SMTP Host`	Name or IP address of the email server. Type: string Use: Required Default: None (required parameter)
`Address Column`	The name of the column where a semicolon-delimited list of recipient email addresses is stored. Type: string Use: Required Default: None (required parameter)

`CC Column`	The name of the column where a semicolon-delimited list of recipient Cc addresses is stored. Type: string Use: Advanced Default: No Cc emails will be sent
`BCC Column`	The name of the column where a semicolon-delimited list of recipient Bcc addresses is stored. Type: string Use: Advanced Default: No Bcc emails will be sent
`Column Names`	Colon-delimited names of stream columns whose values will be included in the email. Type: string Use: Advanced Default: The email will contain values of all columns in the stream.
`Show Column Names`	If true, column names will be included in the email along with their values. If false, only the values will be included. Type: boolean Use: Advanced Default: True
`from`	Email address of the sender. Type: string Use: Required Default: None (required parameter)
`Importance Column`	Name of the stream column where the email importance is stored. Valid values are: high, normal, and low. The default value is “normal”. The values are case-sensitive. Type: string Use: Required Default: None (required parameter)
`Subject Column`	Name of the stream column where the email subject is stored. Type: string Use: Required

	Default: None (required parameter)
`Number of Resend Attempts`	The number of times to retry sending an email if the initial attempt to send it fails. Type: Unsigned integer Use: Advanced Default: 0 (no attempt is made to resend emails) Choose a moderate value (0 - 10) for this parameter. Requiring a large number of attempts to resend the email may lead to excessive memory consumption, particularly if aggravated by network problems and a high volume of records waiting to be emailed.
`Log Alert`	If true, logs an alert at debug level 1 each time the email sending has been successful or failed. Type: boolean Use: Advanced Default: True
`Date Format`	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- Microsoft Outlook® users must disable the feature that removes extra line breaks as follows:
  1. Open Outlook, go to the Tools menu and click **Options.**
  2. On the Preferences tab, click the **E-mail Options** button.
  3. Click to clear the **Remove extra line breaks in plain text messages** check box. Click **OK** twice.

F.34. Sample Plug-in Connector XML File Input

An example of the plug-in connector framework that reads the data from AleriML files by calling the command-line tools **sp_convert and sp_upload.**

Parameters

`File`	File to upload Use: Required
`User`	User name to connect to the platform. Use: Optional
`Password`	Password to connect to the platform. Use: Optional
`Date Format`	Format string for parsing date values. Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- For advanced users with extensive programming expertise.
- The supported parameters are simplistic.

F.35. Sample Plug-in Connector XML File Output

An example of the plug-in connector framework that writes a stream's data to AleriML files by using the command line tool **sp_subscribe.**

Parameters

`File`	File to write to Use: Required
`User`	User name to connect to the platform. Use: Optional
`Password`	Password to connect to the platform. Use: Optional
`Date Format`	Format string for parsing date values. Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Include Base Content`	Start by recording the initial contents of the stream, not just the updates. Type: boolean

	Use: Optional Default: False
Only Base Content	Send only the snapshot of initial contents of the stream, once. Type: boolean Use: Optional Default: False

Known Limitations:

- For advanced users with extensive programming expertise.
- The supported parameters are simplistic.

F.36. Socket (As Client) CSV Input

Receives data in Aleri delimited format from the outgoing network connectors. The connector initiates the connection to another program, then another program sends the data. The data might not have the header, (same as accepted by **sp_convert**), or with the header specifying the field names.

Parameters

Delimiter	Symbol used to separate the columns. Use: Advanced Default: Comma `(,)`
Has Header	Whether the first line of the file contains the description of the fields. Type: boolean Use: Advanced Default: False
Server	Server host name Use: Required
Port	Server port, or -1 to read from the Ephemeral Port File (see advanced parameters). Type: int Use: Required
Ephemeral Port File	File that will contain the server port number, if Port is -1. Use: Advanced

Retry Period (seconds)	Period for trying to re-establish an outgoing connection, in seconds. Type: uint Use: Advanced Default: 1
Enter Initial State	When the connector enters the initial loading state. Use: Advanced Default: Never
Convert to Safe Opcodes	It converts the opcodes INSERT and UPDATE to UPSERT, DELETE to SAFEDELETE. Type: boolean Use: Advanced Default: False
Skip Deletes	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
Date Format	Format string for parsing date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Format string for parsing timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
expectStreamNameOpcode	If true, the first two fields are interpreted as stream name and Aleri op code respectively. Messages with unmatched stream names are discarded. Type: boolean Use: Optional Default: False
Block Size	Number of records to block into one pseudo-transaction Type: int

	Use: Advanced Default: 1
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- The stream name in the file rows is ignored.
- All the data is sent to the same stream.

F.37. Socket (As Client) CSV Output

Send data in Aleri's delimited format to the outgoing network. The connector initiates the connection to another program and then sends the data. The data might not have the header (same as accepted by (sp_convert), or with the header specifying the field names. If the connection is broken, the connector can retry it.

This connector can now be configured to send only the base state of the stream. It sends the data once and exits, but it can be restarted later.

Parameters

`Server`	Server host name Use: Required
`Port`	Server port, or -1 to read from the Ephemeral Port File. Type: int Use: Required
`Ephemeral Port File`	File that will contain the server port number, if port is -1. Use: Advanced
`Retry Period (seconds)`	Period for trying to re-establish an outgoing connection, in seconds. Type: uint Use: Advanced Default: 1
`Include Base Content`	Start by recording the initial contents of the stream, not just the updates. Type: boolean

	Use: Optional Default: False
`Only Base Content`	It sends only the initial contents of the stream, once. Type: boolean Use: Advanced Default: False
`Delimiter`	Symbol used to separate the columns. Use: Advanced Default: Comma `(,)`
`Has Header`	Whether the first line of the file contains the description of the fields. Type: boolean Use: Advanced Default: False
`Date Format`	Format string to parse date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`expectStreamNameOpcode`	If true, the first two fields are interpreted as stream name and Aleri op code respectively. Messages with unmatched stream names are discarded. Type: boolean Use: Optional Default: False
`Prepend StreamNameOpcode`	If true, each message will start with the stream name and the Aleri op code. Type: boolean Use: Optional

	Default: False
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

F.38. Socket (As Client) XML Input

Receives data in AleriML format from the outgoing network connectors. The connector initiates the connection to another program, then another program sends the data. The data might not have the header, (same as accepted by **sp_convert**), or with the header specifying the field names.

Parameters

`Server`	Server host name Use: Required
`Port`	Server port, or -1 to read from the Ephemeral Port File. Type: int Use: Required
`Match stream name`	If true, the XML element name will be matched against the stream name. Unmatched messages will be discarded. Type: boolean Use: Optional Default: False
`Ephemeral Port File`	File that will contain the server port number, if port is -1. Use: Advanced
`Retry Period (seconds)`	Period for trying to re-establish an outgoing connection, in seconds. Type: uint Use: Advanced Default: 1
`Enter Initial State`	When the connector enters the initial loading state. Use: Advanced Default: Never
`Convert to Safe Opcodes`	It converts the opcodes INSERT and UPDATE to UPSERT, DELETE to SAFEDELETE. Type: boolean

	Use: Advanced
	Default: False
`Skip Deletes`	It skips the rows with opcodes DELETE or SAFEDELETE.
	Type: boolean
	Use: Advanced
	Default: False
`Date Format`	Format string to parse date values
	Type: string
	Use: Advanced
	Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string to parse timestamp values
	Type:String
	Use: Advanced
	Default: %Y-%m-%dT%H:%M:%S
`Block Size`	Number of records to block into one pseudo-transaction
	Type: int
	Use: Advanced
	Default: 1
`Field Mapping`	Mapping between the in-platform and external fields
	Use: Advanced

F.39. Socket (As Client) XML Output

Sends data in AleriML format to the outgoing network connections; the connector initiates the connection to another program and then sends the data. If the connection is broken, the connector can retry it.

This connector can now be configured to send only the base state of the stream. It sends the data once and exits, but it can be restarted later.

Parameters

`Server`	Server host name
	Use: Required
`Port`	Server port, or -1 to read from the Ephemeral Port File.
	Type: int

	Use: Required
`Ephemeral Port File`	File that will contain the server port number, if port is -1. Use: Advanced
`Retry Period, s`	Period for trying to re-establish an outgoing connection, in seconds. Type: uint Use: Advanced Default: 1
`Include Base Content`	Start by recording the initial contents of the stream, not just the updates. Type: boolean Use: Optional Default: False
`Only Base Content`	It sends only the initial contents of the stream, once. Type: boolean Use: Advanced Default: False
`Date Format`	Format string to parse date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

F.40. Socket (As Server) XML Input

Receives data in the AleriML format from the incoming network connections. Another program initiates the connection and then sends the data.

This connector can be configured to send only the the base state of the stream but may be repeatedly re-connected.

Parameters

Port	Port number to listen on, or -1 for an 'ephemeral' port (see advanced parameters). Type: int Use: Required
Ephemeral Port File	File where the automatically selected ephemeral port number will be written, if Port is -1. Use: Advanced
Initial Listen Period (seconds)	How long to wait for the first incoming connection before switching to the continuous state. Type: uint Use: Advanced Default: 0
Enter Initial State	When the connector enters the initial loading state. Use: Advanced Default: Never
Convert to Safe Opcodes	It converts the opcodes INSERT and UPDATE to UPSERT, DELETE to SAFEDELETE. Type: boolean Use: Advanced Default: False
Skip Deletes	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
Date Format	Format string for parsing date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Format string for parsing timestamp values Type: String

	Use: Advanced Default: %Y-%m-%dT%H:%M:%S
matchStreamName	If true, the XML element name will be matched against the stream name. Unmatched messages will be discarded. Type: boolean Use: Optional Default: False
Block Size	Number of records to block into one pseudo-transaction Type: int Use: Advanced Default: 1
Field Mapping	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- The stream's name in the file entries is ignored.
- All the data is sent to the same stream.
- Supports only one network connection at a time.

F.41. Socket (As Server) XML Output

Receives data in the AleriML format from the outgoing network connections. Another program initiates the connection and then receives the data.

This connector can be configured to send only the base state of the stream. The socket closes after sending the base state of the stream but may be repeatedly reconnected.

Parameters

Port	Port number to listen on, or -1 for an 'ephemeral' port. Type: int Use: Required
Ephemeral Port File	File where the automatically selected ephemeral port number will be written, if port is -1. Use: Advanced
Include Base Content	Start by recording the initial contents of the stream, not just the updates.

	Type: boolean Use: Optional Default: False
`Only Base Content`	It sends only the initial contents of the stream, once. Type: boolean Use: Advanced Default: False
`Date Format`	Format string to parse date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- Supports only one network connection at a time.

F.42. Socket (As Server) CSV Input

Receives data in Aleri delimited format from the incoming network connections. Another program initiates the connection and then sends the data. The data might not have a header (same as accepted by (sp_convert), or with a header specifying the field names.

Parameters

`Delimiter`	Symbol used to separate the columns. Use: Advanced Default: Comma `(,)`
`Has Header`	Whether the first line of the file contains the description of the fields.

	Type: boolean Use: Advanced Default: False
Port	Port number to listen on, or -1 for an 'ephemeral' port (see advanced parameters). Type: int Use: Required
Ephemeral Port File	File where the automatically selected ephemeral port number will be written, if port is -1. Use: Advanced
Initial Listen Period (seconds)	How long to wait for the first incoming connection before switching to the continuous state. Type: uint Use: Advanced Default: 0
Enter Initial State	When the connector enters the initial loading state. Use: Advanced Default: Never
Convert to Safe Opcodes	It converts the opcodes INSERT and UPDATE to UPSERT, DELETE to SAFEDELETE. Type: boolean Use: Advanced Default: False
Skip Deletes	It skips the rows with opcodes DELETE or SAFEDELETE. Type: boolean Use: Advanced Default: False
Date Format	Format string for parsing date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Format string for parsing timestamp values

	Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
expectStreamNameOpcode	If true, the first two fields are interpreted as stream name and Aleri op code respectively. Messages with unmatched stream names are discarded. Type: boolean Use: Optional Default: False
Block Size	Number of records to block into one pseudo-transaction Type: int Use: Advanced Default: 1
Field Mapping	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- The stream name in the file rows is ignored.
- All the data is sent to the same stream.
- Supports only one network connection.

F.43. Socket (As Server) CSV Output

Sends data in Aleri delimited format from the incoming network connections. Another program initiates the connection and then receives the data. The data might not have the header (same as accepted by **sp_convert**), or with the header specifying the field names.

This connector can be configured to send only the base state of the stream. The socket closes after sending the base state of the stream but may be repeatedly reconnected.

Parameters

Port	Port number to listen on, or -1 for an 'ephemeral' port. Type: int Use: Required
Ephemeral Port File	File where the automatically selected ephemeral port number will

	be written, if port is -1. Use: Advanced
`Include Base Content`	Start by recording the initial contents of the stream, not just the updates. Type: boolean Use: Optional Default: False
`Only Base Content`	It sends only the initial contents of the stream, once. Type: boolean Use: Advanced Default: False
`Delimiter`	Symbol used to separate the columns. Use: Advanced Default: Comma `(,)`
`Has Header`	Whether the first line of the file contains the description of the fields. Type: boolean Use: Advanced Default: False
`Date Format`	Format string to parse date values Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Format string to parse timestamp values Type:String Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Prepend StreamNameOpcode`	If true, each message will start with the stream name and the Aleri op code. Type: boolean Use: Optional Default: False

`Field Mapping`	Mapping between the in-platform and external fields Use: Advanced

Known Limitations:

- Supports only one network connection at a time.

F.44. Socket FIX Input

Reads FIX messages from a TCP server socket and writes them as stream records. Each stream hosts FIX messages of a certain type. Messages of any other FIX type are discarded. All FIX fields except the following are stored in the same order in stream columns:

- BeginString
- BodyLength
- MsgType
- CheckSum

The names of the stream columns must correspond to the FIX protocol specification.

Parameters

`FIX Version`	Version of the FIX protocol. Type: choice Use: Required Default: 4.2
`FIX Message Type`	The type of messages hosted by the stream Type: string Use: Required Default: None (required parameter)
`FIX Host`	Name or IP address of source server Type: string Use: Required Default: None (required parameter)
`Destination Port`	Port on which the messages are available on. Type: Unsigned integer Use: Required

	Default: None (required parameter)
Reconnect Interval	Reconnect interval in seconds. If 0, makes no attempt to reconnect. Type: Unsigned integer Use: Required Default: 10
Maximum Reconnect Attempts	Maximum number of reconnect attempts. Type: Unsigned integer Use: Required Default: 0
Date Format	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
Timestamp Format	Timestamp format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- **This connector is not a full FIX Engine**. If you require a full FIX Engine, please contact a Sybase sales representative for information about the standalone Aleri FIX Engine connector version 1.0.
- FIX versions 4.2 and 4.3 are supported only.
- Repeating groups and components are not supported.
- Only supports insert Opcode.

F.45. Socket FIX Output

Writes stream data as FIX messages to a TCP server socket. Each stream hosts FIX messages of a certain type. Messages are sent contiguously, with no line feeds. The following FIX fields are generated by the connector:

- BeginString
- BodyLength

- MsgType
- CheckSum

The rest of the fields must be stored in the appropriate order in stream columns. The names of the stream columns must correspond to the FIX protocol specification.

Parameters

`FIX Version`	Version of the FIX protocol Type: choice Use: Required Default: 4.2
`FIX Message Type`	The type of messages hosted by the stream Type: string Use: Required Default: None (required parameter)
`Destination Host`	Name or IP address of the destination Type: string Use: Required Default: None (required parameter)
`Destination Port`	Port on which the server socket is listening to messages. Type: Unsigned integer Use: Required Default: None (required parameter)
`Date Format`	Date format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S
`Timestamp Format`	Timestamp format Type: string Use: Advanced Default: %Y-%m-%dT%H:%M:%S

Known Limitations:

- **This connector is not a full FIX Engine**. If you require a full FIX Engine, please contact a Sybase sales representative for information about the standalone Aleri FIX connector version 1.0.
- Only versions 4.2 and 4.3 of FIX are supported
- Repeating groups and components are not supported
- No reconnection attempt is made if the connection to the FIX server is lost
- Only supports insert Opcode.

F.46. SybaseIQ Output

An external connector that starts the sp_archive utility and loads data from the Sybase Aleri Streaming Platform into Sybase IQ. It uses SybaseIQ's bulk load feature to efficiently load insert only data and automatically switches to use the slower ODBC® mechanism when it encounters updates and deletes. This connector is designed to be highly robust. It caches all the data on disk and also uses the persistent subscribe mechanism, which ensures no data is lost even if the connection to the the Sybase Aleri Streaming Platform is lost.

Parameters

User Name	The user name for connecting to the Sybase Aleri Streaming Platform. This parameter is required when the Sybase Aleri Streaming Platform uses an authentication type of RSA, Kerberos, or PAM (when the underlying authentication mechanism for PAM requires User and Password). Type: string Use: Required Default: None
Password	An optional parameter required only when the Sybase Aleri Streaming Platform uses the PAM authentication type that has an underlying authentication mechanism with User and Password. Type: string Use: Optional Default: None
RSA Key File	The full path to the RSA Key private key file This parameter is required if the Sybase Aleri Streaming Platform uses the RSA authentication mechanism. Type: string Use: Optional Default: none
Use Kerberos	A Boolean value that needs to be set to true when the Sybase Aleri Streaming Platform is configured to use Kerberos authentication. If this is set to true, the User Name parameter must be provided.

Type: boolean

Use: Optional

Default: False

`Configuration File`

The full path to the configuration file that provides Sybase IQ connection details and load options.

Type: string

Use: Required

Default: None (required parameter)

`Archive Deltas`

Archives deltas when set to true otherwise only a snapshot of the data is archived.

Type: boolean

Use: Advanced

Default: True

`Swap Bytes`

Set to true when the Platform and the archive utility are running on different architectures(Little/Big Endian advanced).

Type: boolean

Use: Advanced

Default: False

`Recover Only`

Recovers any data that was read from the Sybase Aleri Streaming Platform but not archived, and exits.

Type: boolean

Use: Advanced

Default: False

`Datawarehousing Mode`

Any updates are treated as inserts and deletes are ignored when set to true. Use this mode when you want to archive the data for historical purposes.

Type: boolean

Use: Advanced

Default: False

`Archive Interval`

Specifies how long to wait in seconds after each time a set of data is archived and the next set of data is archived.

Type: uint

Use: Advanced

	Default: 1
`Precision`	A value between 0 and 6 that specifies the precision to use for money and float datatypes Type: uint Use: Advanced Default: 6
`Commit Batch Size`	This is the commit batch size used when using SQL to archive the data instead of the bulk load mechanism. For bulk loads specify the batch size in the configuration file Type: uint Use: Advanced Default: 1000
`ODBC Retry Attempts`	The number of times to retry the ODBC connection if a connection cannot be made Type: uint Use: Advanced Default: 5
`ODBC Retry Interval`	The number of seconds to wait before retrying the ODBC connection. Type: uint Use: Advanced Default: 60

Known Limitations:

- Attaching a SybaseIQOut connector to a stream does not guarantee that the data from this stream will be archived. The connector uses the specified configuration file to get this information. If a different stream is specified by the configuration file then that one will be archived.

F.47. Teradata Output

An external connector that starts the Teradata TPump utility and loads data from the Sybase Aleri Streaming Platform into Teradata. The AleriTeradata connector for the TPump access module is designed to load data in a robust, efficient and a lossless manner even if the Sybase Aleri Streaming Platform or Teradata connection is lost.

Each connector is designed to do inserts, updates and deletes into a single table in the Teradata database from a single stream in the Sybase Aleri Streaming Platform. There can be more than one connector associated with a single stream.

The persistent subscribe mechanism is used to make the connector robust and lossless. That means there needs to be two additional streams to associate with each Teradata output connector and stream combination.

The first stream (named StreamName_log by default) contains a log of every event that affected the outputted stream. The second stream (usually called Stream- Name_control) purges the transactions in StreamName_log after the information has been committed to the Teradata database.

The heart of the connector is the user-defined TPump script file, which is highly configurable and can filter or modify data received from the Sybase Aleri Streaming Platform. Two sample script files are included in the folder INSTALL DIRECTORY/examples/scripts/teradata. These examples archive data produced by the VWAP example model in the same location. Teradata has additional documentation on its website about the TPump Access Module.

Parameters

`TPump Script File`	The full path and name of the TPump script file. Type: string Use: Required Default: None (required parameter)
`TPump Executable`	The full path and name of the TPump executable. If not provided, the tpump executable directory must be in the `PATH`. The default executable name is `tpump.exe` for Windows and `tpumpexe` for Linux/Solaris. Type: string Use: Optional Default: None (required parameter)

Access Module Options

Access Module Options	The AleriTeradata access module takes many arguments. These arguments are specified in the Tpump script file using the following syntax, **IMPORT INFILE MYFILE AXSMOD <INSTALL_DIR>/lib/AleriTeradata.so 'option1 option1Val option2 option2Val..'** The options that can be specified are as follows

Parameters, Properties, Options

`-B batchsize`	This parameter specifies the number of records to pack in each block before sending it to Teradata and controls the internal buffer size. It should usually be set to the same value as the PACK property in the Tpump script file.
`-c UserID:PassWd`	This parameter is required when the platform uses PAM, RSA, or Kerberos authentication. The UserId component is required when spe-

	cified. The Password component is only required when using PAM authentication and the underlying authentication mechanism has User and Password.
`-e`	If this property is specified, it means that the communication with the Sybase Aleri Streaming Platform is encrypted. The Sybase Aleri Streaming Platform must be started in encrypted mode in order to use this property,
`-G`	This property specifies that Kerberos must be used as the authentication mechanism. If this option is specified then the **-c** option must also be specified.
`-Host[hostname:]port:`	This option is required when the Sybase Aleri Streaming Platform is configured to use hot spares and it specifies the hostname that the Hot Spare is running on and the command and control port. While the port is required. the hostname component is optional and defaults to localhost if not provided.
`-p[hostname:]port`	This option specifies the hostname the Sybase Aleri Streaming Platform is running on and the Command and Control port. The port is required, but the hostname component is optional and defaults to 'localhost' if not provided.
`-I`	This is an optional parameter specifying an insert-only mode. In this mode, all updates are treated as inserts and all deletes are ignored.
`-k rsaKeyFile`	If this option is specified, RSA authentication is used to connect to the Sybase Aleri Streaming Platform. The rsaKeyFile is the path and file name of the RSA private key file.
`-R TruncateStream`	This required property is used to specify the StreamName_truncate name described below.
`-s StreamName`	Every stream with data that needs to be exported to Teradata must be associated with a persistent subscribe pattern. The persistent subscribe pat-

	tern ensures there is no loss of data when the Sybase Aleri Streaming Platform or the Tpump executable is stopped and restarted. Each persistent subscribe pattern adds two extra streams to the model. The streams are named by default, StreamName_log and StreamName_truncate. For example, if the stream to export to Teradata is “Trades,” then the persistent subscribe pattern will add two streams: “Trades_log” and “Trades_truncate”. This required property is used to specify the StreamName_log stream name, such as Trades_log in the above example.
`-S`	If this option is specified, a snapshot of the specified stream is sent to the Sybase Aleri Streaming Platform and the connector exits.
`-T Interval`	This interval specifies how often to generate a dummy message if there is no data so that the Teradata database can check point any data that hasn't been check pointed. The default is every 10 seconds if it's not specified.

Known Limitations:

- The connector does not work across different architectures. For example, if the Teradata database is running on a Big Endian architecture such as a Sun SPARC® and the connector is run on a Little Endian architecture such as an Intel® machine or vice versa, the connector will not work.

- Due to the lossless nature of the connector there may be some duplicate entries that need to be handled on a restart. To ensure that this is handled properly, the IGNORE DUPLICATES option must be specified in the TPump script file.

- Although the TPump and the configuration script file is designed to handle loading multiple tables, the AleriTeradata access module is not designed to handle this. Therefore, each script file can be used to process only one source stream and destination table.

F.48. Tibco Rendezvous Plug-in

The Tibco Rendezvous Plug-in connector publishes stream data to a Rendezvous subject and vice-versa. It can be configured on any source stream as an inbound data location. The authentication method is set to that of the Sybase Aleri Streaming Platform: none, pam, rsa, or gssapi.

The Aleri Tibco Rendezvous Adapter version 1.0 or later must be installed to use this Plug-in. Please contact your Sybase sales representative for more information about the Adapter.

Parameters

`Connector Directory Path`	Specify the absolute path to the Adapter's installation directory. This parameter is ignored if the *`Connector Remote Directory Path`* parameter is supplied. Type: directory Use: Required Default: None
`Configuration File Path`	Specify the absolute path to the Adapter's configuration file. This parameter is ignored if the *`Remote Configuration File Path`* parameter is supplied. Type: configFilename Use: Required Default: None
`Connector Remote Directory Path`	Specify the path to the connector remote base directory (for remote execution only). If this parameter is supplied, the *`Connector Directory Path`* parameter is ignored. Type: string Use: Advanced Default: None
`Remote Configuration File Path`	Specify the path to the connector's remote configuration file (for remote execution only). If this parameter is supplied, the *`Configuration File Path`* parameter is ignored. Type: string Use: Advanced Default: None

F.49. Wombat Plug-in

The Wombat Plug-in connects to a Wombat data feed to receive real-time Level 1 and Level 2 market data. It can be configured on any source stream as an inbound data location. The authentication method is set to that of the Sybase Aleri Streaming Platform: none, pam, rsa, or gssapi. This connector supports discovery.

The Aleri Wombat Adapter version 1 or later must be installed to use this Plug-in. You can refer to Aleri Wombat Adapter documentation for installation and configuration details. Your Sybase sales representative can provide more details.

Parameters

`Connector Directory Path`	Specify the absolute path to the Adapter's installation directory. This parameter is ignored if the *`Connector Remote Directory Path`* parameter is supplied. Type: directory Use: Required Default: None
`Configuration File Path`	Specify the absolute path to the Adapter's configuration file. This parameter is ignored if the *`Remote Configuration File Path`* parameter is supplied. Type: configFilename Use: Required Default: None
`Discovery Directory Path`	Specify the absolute path to the Adapter's discovery directory. Type: directory Use: Required Default: None
`Connector Remote Directory Path`	Specify the path to the connector remote base directory (for remote execution only). If this parameter is supplied, the *`Connector Directory Path`* parameter is ignored. Type: string Use: Advanced Default: None
`Remote Configuration File Path`	Specify the path to the connector's remote configuration file (for remote execution only). If this parameter is supplied, the *`Configuration File Path`* parameter is ignored. Type: string Use: Advanced Default: None

Appendix G. List of Time Zones

Below is a list of time zones used in the Sybase Aleri Streaming Platform from the industry-standard Olson timezone (also known as TZ) database. Note that there is no value for the abbreviation "PST" for Pacific Standard Time. Use "PST8PDT" for the time zone that takes into account daylight savings time.

ACT	AET	AGT
ART	AST	Africa/Abidjan
Africa/Accra	Africa/Addis_Ababa	Africa/Algiers
Africa/Asmera	Africa/Bamako	Africa/Bangui
Africa/Banjul	Africa/Bissau	Africa/Blantyre
Africa/Brazzaville	Africa/Bujumbura	Africa/Cairo
Africa/Casablanca	Africa/Ceuta	Africa/Conakry
Africa/Dakar	Africa/Dar_es_Salaam	Africa/Djibouti
Africa/Douala	Africa/El_Aaiun	Africa/Freetown
Africa/Gaborone	Africa/Harare	Africa/Johannesburg
Africa/Kampala	Africa/Khartoum	Africa/Kigali
Africa/Kinshasa	Africa/Lagos	Africa/Libreville
Africa/Lome	Africa/Luanda	Africa/Lubumbashi
Africa/Lusaka	Africa/Malabo	Africa/Maputo
Africa/Maseru	Africa/Mbabane	Africa/Mogadishu
Africa/Monrovia	Africa/Nairobi	Africa/Ndjamena
Africa/Niamey	Africa/Nouakchott	Africa/Ouagadougou
Africa/Porto-Novo	Africa/Sao_Tome	Africa/Timbuktu
Africa/Tripoli	Africa/Tunis	Africa/Windhoek
America/Adak	America/Anchorage	America/Anguilla
America/Antigua	America/Araguaina	America/Argentina/Buenos_Aires
America/Argentina/Catamarca	America/Argen-tina/ComodRivadavia	America/Argentina/Cordoba
America/Argentina/Jujuy	America/Argentina/La_Rioja	America/Argentina/Mendoza
America/Argentina/Rio_Gallegos	America/Argentina/San_Juan	America/Argentina/Tucuman
America/Argentina/Ushuaia	America/Aruba	America/Asuncion
America/Atka	America/Bahia	America/Barbados
America/Belem	America/Belize	America/Boa_Vista
America/Bogota	America/Boise	America/Buenos_Aires
America/Cambridge_Bay	America/Campo_Grande	America/Cancun
America/Caracas	America/Catamarca	America/Cayenne
America/Cayman	America/Chicago	America/Chihuahua
America/Coral_Harbour	America/Cordoba	America/Costa_Rica
America/Cuiaba	America/Curacao	America/Danmarkshavn
America/Dawson	America/Dawson_Creek	America/Denver
America/Detroit	America/Dominica	America/Edmonton
America/Eirunepe	America/El_Salvador	America/Ensenada
America/Fort_Wayne	America/Fortaleza	America/Glace_Bay
America/Godthab	America/Goose_Bay	America/Grand_Turk
America/Grenada	America/Guadeloupe	America/Guatemala
America/Guayaquil	America/Guyana	America/Halifax
America/Havana	America/Hermosillo	America/Indiana/Indianapolis
America/Indiana/Knox	America/Indiana/Marengo	America/Indiana/Petersburg
America/Indiana/Vevay	America/Indiana/Vincennes	America/Indianapolis
America/Inuvik	America/Iqaluit	America/Jamaica
America/Jujuy	America/Juneau	America/Kentucky/Louisville
America/Kentucky/Monticello	America/Knox_IN	America/La_Paz
America/Lima	America/Los_Angeles	America/Louisville
America/Maceio	America/Managua	America/Manaus

America/Martinique	America/Mazatlan	America/Mendoza
America/Menominee	America/Merida	America/Mexico_City
America/Miquelon	America/Moncton	America/Monterrey
America/Montevideo	America/Montreal	America/Montserrat
America/Nassau	America/New_York	America/Nipigon
America/Nome	America/Noronha	America/North_Dakota/Center
America/Panama	America/Pangnirtung	America/Paramaribo
America/Phoenix	America/Port-au-Prince	America/Port_of_Spain
America/Porto_Acre	America/Porto_Velho	America/Puerto_Rico
America/Rainy_River	America/Rankin_Inlet	America/Recife
America/Regina	America/Rio_Branco	America/Rosario
America/Santiago	America/Santo_Domingo	America/Sao_Paulo
America/Scoresbysund	America/Shiprock	America/St_Johns
America/St_Kitts	America/St_Lucia	America/St_Thomas
America/St_Vincent	America/Swift_Current	America/Tegucigalpa
America/Thule	America/Thunder_Bay	America/Tijuana
America/Toronto	America/Tortola	America/Vancouver
America/Virgin	America/Whitehorse	America/Winnipeg
America/Yakutat	America/Yellowknife	Antarctica/Casey
Antarctica/Davis	Antarctica/DumontDUrville	Antarctica/Mawson
Antarctica/McMurdo	Antarctica/Palmer	Antarctica/Rothera
Antarctica/South_Pole	Antarctica/Syowa	Antarctica/Vostok
Arctic/Longyearbyen	Asia/Aden	Asia/Almaty
Asia/Amman	Asia/Anadyr	Asia/Aqtau
Asia/Aqtobe	Asia/Ashgabat	Asia/Ashkhabad
Asia/Baghdad	Asia/Bahrain	Asia/Baku
Asia/Bangkok	Asia/Beirut	Asia/Bishkek
Asia/Brunei	Asia/Calcutta	Asia/Choibalsan
Asia/Chongqing	Asia/Chungking	Asia/Colombo
Asia/Dacca	Asia/Damascus	Asia/Dhaka
Asia/Dili	Asia/Dubai	Asia/Dushanbe
Asia/Gaza	Asia/Harbin	Asia/Hong_Kong
Asia/Hovd	Asia/Irkutsk	Asia/Istanbul
Asia/Jakarta	Asia/Jayapura	Asia/Jerusalem
Asia/Kabul	Asia/Kamchatka	Asia/Karachi
Asia/Kashgar	Asia/Katmandu	Asia/Krasnoyarsk
Asia/Kuala_Lumpur	Asia/Kuching	Asia/Kuwait
Asia/Macao	Asia/Macau	Asia/Magadan
Asia/Makassar	Asia/Manila	Asia/Muscat
Asia/Nicosia	Asia/Novosibirsk	Asia/Omsk
Asia/Oral	Asia/Phnom_Penh	Asia/Pontianak
Asia/Pyongyang	Asia/Qatar	Asia/Qyzylorda
Asia/Rangoon	Asia/Riyadh	Asia/Riyadh87
Asia/Riyadh88	Asia/Riyadh89	Asia/Saigon
Asia/Sakhalin	Asia/Samarkand	Asia/Seoul
Asia/Shanghai	Asia/Singapore	Asia/Taipei
Asia/Tashkent	Asia/Tbilisi	Asia/Tehran
Asia/Tel_Aviv	Asia/Thimbu	Asia/Thimphu
Asia/Tokyo	Asia/Ujung_Pandang	Asia/Ulaanbaatar
Asia/Ulan_Bator	Asia/Urumqi	Asia/Vientiane
Asia/Vladivostok	Asia/Yakutsk	Asia/Yekaterinburg
Asia/Yerevan	Atlantic/Azores	Atlantic/Bermuda
Atlantic/Canary	Atlantic/Cape_Verde	Atlantic/Faeroe
Atlantic/Jan_Mayen	Atlantic/Madeira	Atlantic/Reykjavik
Atlantic/South_Georgia	Atlantic/St_Helena	Atlantic/Stanley
Australia/ACT	Australia/Adelaide	Australia/Brisbane
Australia/Broken_Hill	Australia/Canberra	Australia/Currie

Australia/Darwin	Australia/Hobart	Australia/LHI
Australia/Lindeman	Australia/Lord_Howe	Australia/Melbourne
Australia/NSW	Australia/North	Australia/Perth
Australia/Queensland	Australia/South	Australia/Sydney
Australia/Tasmania	Australia/Victoria	Australia/West
Australia/Yancowinna	BET	BST
Brazil/Acre	Brazil/DeNoronha	Brazil/East
Brazil/West	CAT	CET
CNT	CST	CST6CDT
CTT	Canada/Atlantic	Canada/Central
Canada/East-Saskatchewan	Canada/Eastern	Canada/Mountain
Canada/Newfoundland	Canada/Pacific	Canada/Saskatchewan
Canada/Yukon	Chile/Continental	Chile/EasterIsland
Cuba	EAT	ECT
EET	EST	EST5EDT
Egypt	Eire	Etc/GMT
Etc/GMT+0	Etc/GMT+1	Etc/GMT+10
Etc/GMT+11	Etc/GMT+12	Etc/GMT+2
Etc/GMT+3	Etc/GMT+4	Etc/GMT+5
Etc/GMT+6	Etc/GMT+7	Etc/GMT+8
Etc/GMT+9	Etc/GMT-0	Etc/GMT-1
Etc/GMT-10	Etc/GMT-11	Etc/GMT-12
Etc/GMT-13	Etc/GMT-14	Etc/GMT-2
Etc/GMT-3	Etc/GMT-4	Etc/GMT-5
Etc/GMT-6	Etc/GMT-7	Etc/GMT-8
Etc/GMT-9	Etc/GMT0	Etc/Greenwich
Etc/UCT	Etc/UTC	Etc/Universal
Etc/Zulu	Europe/Amsterdam	Europe/Andorra
Europe/Athens	Europe/Belfast	Europe/Belgrade
Europe/Berlin	Europe/Bratislava	Europe/Brussels
Europe/Bucharest	Europe/Budapest	Europe/Chisinau
Europe/Copenhagen	Europe/Dublin	Europe/Gibraltar
Europe/Helsinki	Europe/Istanbul	Europe/Kaliningrad
Europe/Kiev	Europe/Lisbon	Europe/Ljubljana
Europe/London	Europe/Luxembourg	Europe/Madrid
Europe/Malta	Europe/Mariehamn	Europe/Minsk
Europe/Monaco	Europe/Moscow	Europe/Nicosia
Europe/Oslo	Europe/Paris	Europe/Prague
Europe/Riga	Europe/Rome	Europe/Samara
Europe/San_Marino	Europe/Sarajevo	Europe/Simferopol
Europe/Skopje	Europe/Sofia	Europe/Stockholm
Europe/Tallinn	Europe/Tirane	Europe/Tiraspol
Europe/Uzhgorod	Europe/Vaduz	Europe/Vatican
Europe/Vienna	Europe/Vilnius	Europe/Warsaw
Europe/Zagreb	Europe/Zaporozhye	Europe/Zurich
Factory	GB	GB-Eire
GMT	GMT+0	GMT-0
GMT0	Greenwich	HST
Hongkong	IET	IST
Iceland	Indian/Antananarivo	Indian/Chagos
Indian/Christmas	Indian/Cocos	Indian/Comoro
Indian/Kerguelen	Indian/Mahe	Indian/Maldives
Indian/Mauritius	Indian/Mayotte	Indian/Reunion
Iran	Israel	JST
Jamaica	Japan	Kwajalein
Libya	MET	MIT
MST	MST7MDT	Mexico/BajaNorte

Mexico/BajaSur	Mexico/General	Mideast/Riyadh87
Mideast/Riyadh88	Mideast/Riyadh89	NET
NST	NZ	NZ-CHAT
Navajo	PLT	PNT
PRC	PRT	PST
PST8PDT	Pacific/Apia	Pacific/Auckland
Pacific/Chatham	Pacific/Easter	Pacific/Efate
Pacific/Enderbury	Pacific/Fakaofo	Pacific/Fiji
Pacific/Funafuti	Pacific/Galapagos	Pacific/Gambier
Pacific/Guadalcanal	Pacific/Guam	Pacific/Honolulu
Pacific/Johnston	Pacific/Kiritimati	Pacific/Kosrae
Pacific/Kwajalein	Pacific/Majuro	Pacific/Marquesas
Pacific/Midway	Pacific/Nauru	Pacific/Niue
Pacific/Norfolk	Pacific/Noumea	Pacific/Pago_Pago
Pacific/Palau	Pacific/Pitcairn	Pacific/Ponape
Pacific/Port_Moresby	Pacific/Rarotonga	Pacific/Saipan
Pacific/Samoa	Pacific/Tahiti	Pacific/Tarawa
Pacific/Tongatapu	Pacific/Truk	Pacific/Wake
Pacific/Wallis	Pacific/Yap	Poland
Portugal	ROC	ROK
SST	Singapore	SystemV/AST4
SystemV/AST4ADT	SystemV/CST6	SystemV/CST6CDT
SystemV/EST5	SystemV/EST5EDT	SystemV/HST10
SystemV/MST7	SystemV/MST7MDT	SystemV/PST8
SystemV/PST8PDT	SystemV/YST9	SystemV/YST9YDT
Turkey	UCT	US/Alaska
US/Aleutian	US/Arizona	US/Central
US/East-Indiana	US/Eastern	US/Hawaii
US/Indiana-Starke	US/Michigan	US/Mountain
US/Pacific	US/Pacific-New	US/Samoa
UTC	Universal	VST
W-SU	WET	Zulu

Index